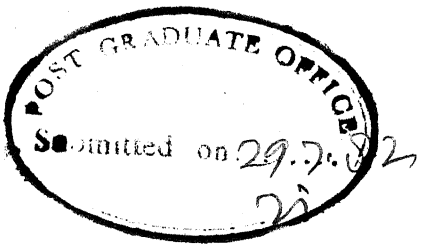
A UNIFORM PROGRAMMING ENVIRONMENT FOR PASCAL-S USERS

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by

S. V. S. NAGESWARA RAO



CERTIFICATE

This is to certify that the work entitled 'A UNIFORM PROGRAMMING ENVIRONMENT FOR PASCAL-S USERS' by S.V.S. Nageswara Rao has been carried out under my supervision and has not been submitted elsewhere for a degree.

H. V. Sahasrabuddhe

Dr. H.V. Sahasrabuddhe
Professor
Computer Science Program
Indian Institute of Technology
Kanpur.

Kanpur
26 July, 1982

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis supervisor Dr. H.V. Sahasrabuddhe for his interest, timely and valuable advice and constant encouragement. His inspiration and guidance is invaluable for this thesis to take shape.

I am also grateful to my brothers Murty and Ranga, friends V.H. Subramanian, Ramabrahmam, Ramagopal and many others for making my stay at I.I.T. Kanpur a memorable experience.

Finally I would like to thank Mr. C.M. Abraham for typing this manuscript excellently.

S.V.S. Nageswara Rao

CONTENTS

CHAPTER 1

INTRODUCTION

'Programs are not text; they are hierarchical compositions of computational structures and should be constructed, edited, compiled/executed and debugged in an environment that consistently acknowledges and reinforces this view point' [TEI 81].

Current programming systems are woefully inadequate in this respect. For example, programs are written using a text editor, which defines a programming environment in terms of lines and characters. Such an editor does not know that the character sequence b,e,g,i,n has a specific meaning. A compiler however defines another environment in which begin is recognized as a significant keyword. From a programmer's viewpoint, it is confusing to have two vastly different environments for typing in a program and compiling/executing it. He has to constantly translate back and forth between the structured object he understands and the sequential form the programming system allows him.

It is therefore desirable to have a programming system which not only preserves the structural form of a program while inputting and storing but also interacts with the programmers at the structural level.

We have implemented such a programming system, based on a source program file organization in which structure is manifest. Recently, Teitelbaum [TEI 81] has announced a syntax oriented programming system, which is also based on program structure. Whereas his is an integrated system, incorporating the functions of an editor, compiler, interpreter and a debugger, ours consists of a program constructor, an incremental compiler and a pretty printing program, all working on a common interface, which is a structured representation of source programs called SSPF (for structured source program files).

The program constructor provides an editing environment, which makes use of the programming language syntax. It is an editor in the sense that it is used for creating or modifying a program. However, its instructions do not operate on lines and characters but on program constructs. Instead of typing the character sequence for the word 'begin', the program constructor provides an instruction, which has the effect of putting the keyword pair begin ... end at the current position of the program text, provided it is syntactically correct.

The chief purpose of the program constructor is to build the structured representation. This structure consists of all the information necessary for generating intermediate code. Problems like non-matching and misspelled key-words

cannot occur.

A major advantage of such a programming environment is that a user cannot write structurally incorrect programs. The SSPF organization allows for a more efficient program translation in that the need for recognition of character sequences as lexical units and lexical units as syntactic units is minimized. It also allows incremental compilation, reducing the burden of scanning the entire source text of program, in case of any modifications of the program. Such a feature is very economical during program development stage.

The pretty printing program maps the structured object (program) into a sequential text, with proper indentation reflecting its structure.

In this thesis, we have described the implementation of a programming system based on SSPF for the programming language Pascal-S. A syntax oriented editor for Pascal-S described in [PAR 81] and an incremental compiler for the same programming language discussed in [JAJ 80] form the basis for our work.

Chapter 2 of this thesis is on structured source program files and also describes how they can be used as a basis for designing a new programming system.

Chapter 3 discusses the format of SSPF for Pascal-S

programs as used in the implementation of the components of our programming system.

Chapter 4 discusses the program constructor and Chapter 5 describes the incremental compiler.

Chapter 6 is a user manual for the programming system.

We conclude our thesis in Chapter 7. The program listings of various components of the programming system have been given as a supplement to this thesis.

CHAPTER 2

STRUCTURED SOURCE PROGRAM FILES

The need for storing programs with their structure preserved has already been stressed. Since a programming system has to identify various computational units in the program to allow modification at that level, storing programs as a text is very inefficient and unnatural.

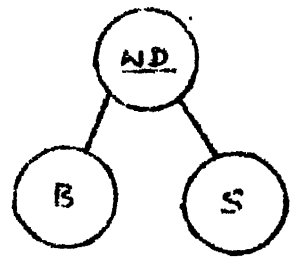
In this chapter we describe how we can store programs with their structure intact, and how such representation can be utilized in a programming system that has the properties, we seek (Chapter 1). Their organization is based on flow-trees suggested in [JAJ 78].

2.1 FLOW TREES

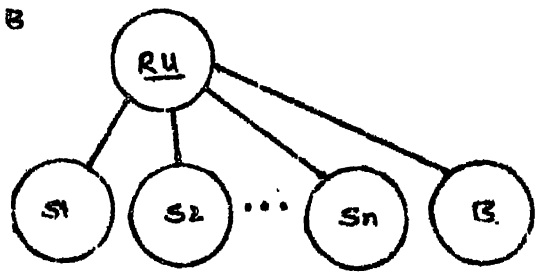
Those constructs which produce a program with nested structure are preferred by many present-day programmers. Such a structure expresses hierarchical relationship among program segments. Flow trees provide a natural way for preserving and displaying this hierarchical relationship. They are useful in understanding the flow of control in terms of the hierarchy implied by the use of various structured constructs.

As the name suggest, flow trees are tree like representations of programs. Each unit in the program is represented

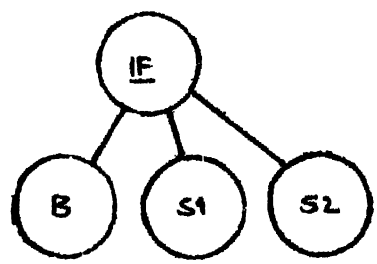
WHILE B DO S



REPEAT S1 ; S2 , ... ; Sn UNTIL B



IF B THEN S1 ELSE S2



FOR <FORLPINX> := <INIT VAL> TO <FIN VAL>
DO S

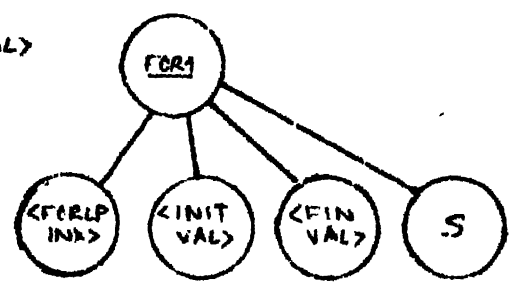


Fig. 3.1 Abstract Segments For various Statements

as a part of the flow tree. Each structured construct has a certain number of components. Such a construct is represented by a node called control-node. Its components are represented by the sons of this node. In each control-node, we can store the type of the control-construct. For example, the control construct 'while B do S' has the structure shown in Fig. 2.1. In the same figure, flow tree representations for other program constructs are also given.

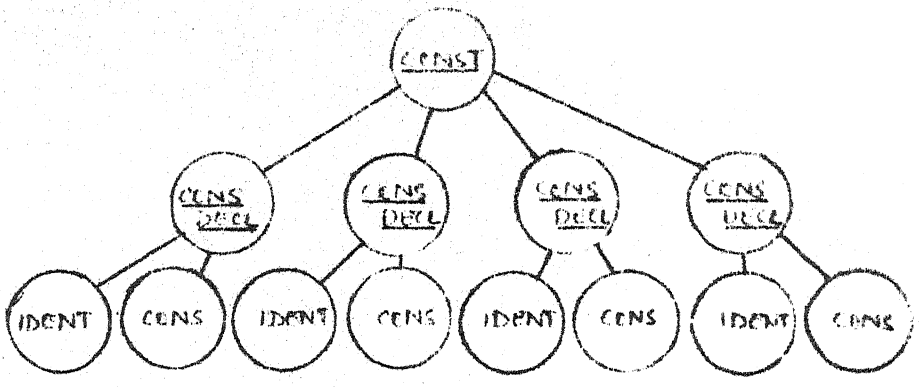
A complete program can be represented in flow tree form, using these basic structures. In flow trees, the leaf (terminal) nodes always contain source program text. It is through the interior (non-terminal) nodes that the structure of the source programs is reflected.

The meaning of any control-node can be derived by identifying the components and knowing the relationship, it imposes on these. This relationship is characterized by the possible execution sequence of the components, which the control node induces. A control node, thus implicitly contains all the information to execute the subtree rooted by it. Consequently, execution control flow can be understood through the traversal of the flow-tree.

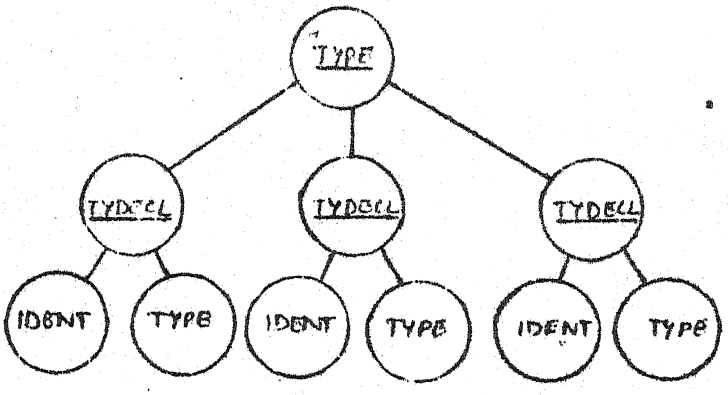
2.2 SSPF ORGANIZATION

We have already emphasized that the programmers interaction with the programming systems should be at a structural level and

CONSTANT DECLARATION



TYPE DECLARATION



VAR DECLARATION

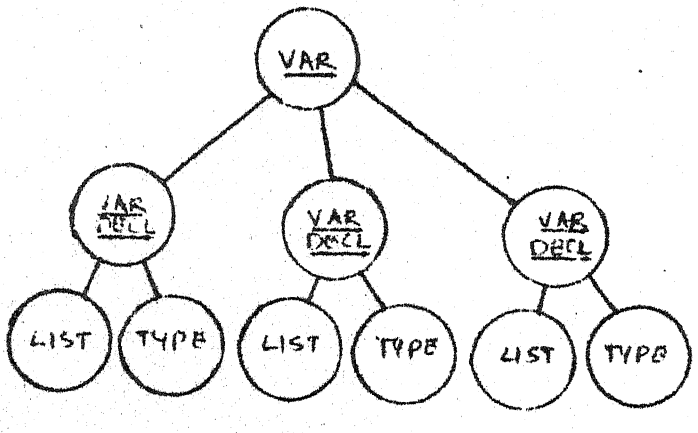


Fig 2.2 Flowtree Segments Showing various Declarations

not at sequential level. Preserving structure in source programs helps in designing a programming systems that allows such interaction. Flow trees provide a framework in which we can design SSPF.

Programs are stored in a tree like fashion with nodes at various levels. At each of these nodes, we can store a great variety of information depending on the type, pointers to sons and possibly also to the parent.

Control-flow information is automatically reflected through the flow-tree representation.

To cater to declaration of identifiers, we can associate the declarations with the root of the subtree over which they are to have effect. We have modified this somewhat, to as to be able to identify various types of declarations (constant, type, variable, procedure and function). These declarations are attached as sons of either prog or proc or func nodes. With a separate node type as a root for each declaration section (see Fig. 2.2).

The exact set of node types used in the SSPF depends on the language used.

2.3 CREATION AND MODIFICATION OF SSPF

When creating the program, the program constructor prompts the user according to the syntax of the programming language and simultaneously maps the source received into

SSPF form. For example, the current node is declared as a wd node. Then the program constructor can respond by creating its two sons and prompting the user to supply a boolean expression for the first son and a statement for the second.

When modifying the program, the program constructor instead of allowing the user to view the program as a sequence of lines, allows him naturally to view it as a composite of several structured units. It can have commands like :

1, Move to the else clause of the current if statement.
2. Move to the next case in the case statement being scanned.
3. Skip over the entire block, that is currently being edited.
4. Move to the program segment named 'FIND'.
5. Substitute one variable name for another in the specified block.
6. Change the specified repeat ... Until construct into a while - do construct.
7. Delete the specified program segment.

2.4 COMPILATION OF SSPF

The SSPF organization outlined above, provides an opportunity for incremental compilation. (If a program is modified, only the modified part need be recompiled). The

compilation proceeds by traversing the SSPF in a fixed order (pre-order) and compiling each node encountered. Compiling each node in the SSPF produces compilation - information which is either used to augment the object code produced thus for or to update the global compiler tables or to do both. This compilation-information is stored in the nodes of the SSPF, as far as possible for use in subsequent compilations to achieve incrementality.

CHAPTER 3

SSPF FORMAT FOR PASCAL-S

We have outlined in Chapter 2, how structure preserving source program files can be used in designing/implementing a new programming environment.

In this chapter, we delineate the format of such files for Pascal-S and how they can be stored in computer such that they can be created and manipulated by the programming system.

One major decision has been where exactly to stop preserving the structure of source programs in a flow-tree form. For example, the node representing an assignment can have two sons; a variable part and an expression part. The expression part can be further broken into factors, terms, identifiers and so on giving rise to a complex flow tree segment. We have decided to stop structuring at statement and declaration level.

3.1 IDENTIFICATION OF NODES IN A FLOW TREE

The program constructor in editing node, allows program modification. The programmer therefore has to be able to identify various program segments and be able to refer to them.

In Cornell-program synthesizer [TEI 81] this is

```
program [illegible](input,output);
  const
    [illegible]='?ABACKWARDS';
  type
    ARRAY=array[...] of integer;
  var
    [illegible]:integer;
    [illegible]:ARRAY;
  procedure [illegible](i:integer);
    var
      [illegible]:integer;
    function [illegible]:integer;
      begin
        while ...<>' ' do
          if ... then
            ...:=...
          else ...:=...
        ...:=...
      end;
    begin
      for ...:=1 to ... do
        begin
          [illegible](X,...);
          [illegible]([illegible])
        end
    end;
  begin
    [illegible]:=...;
    [illegible](1)
  end.
```

Fig. 3.1 Example Pascal/S program

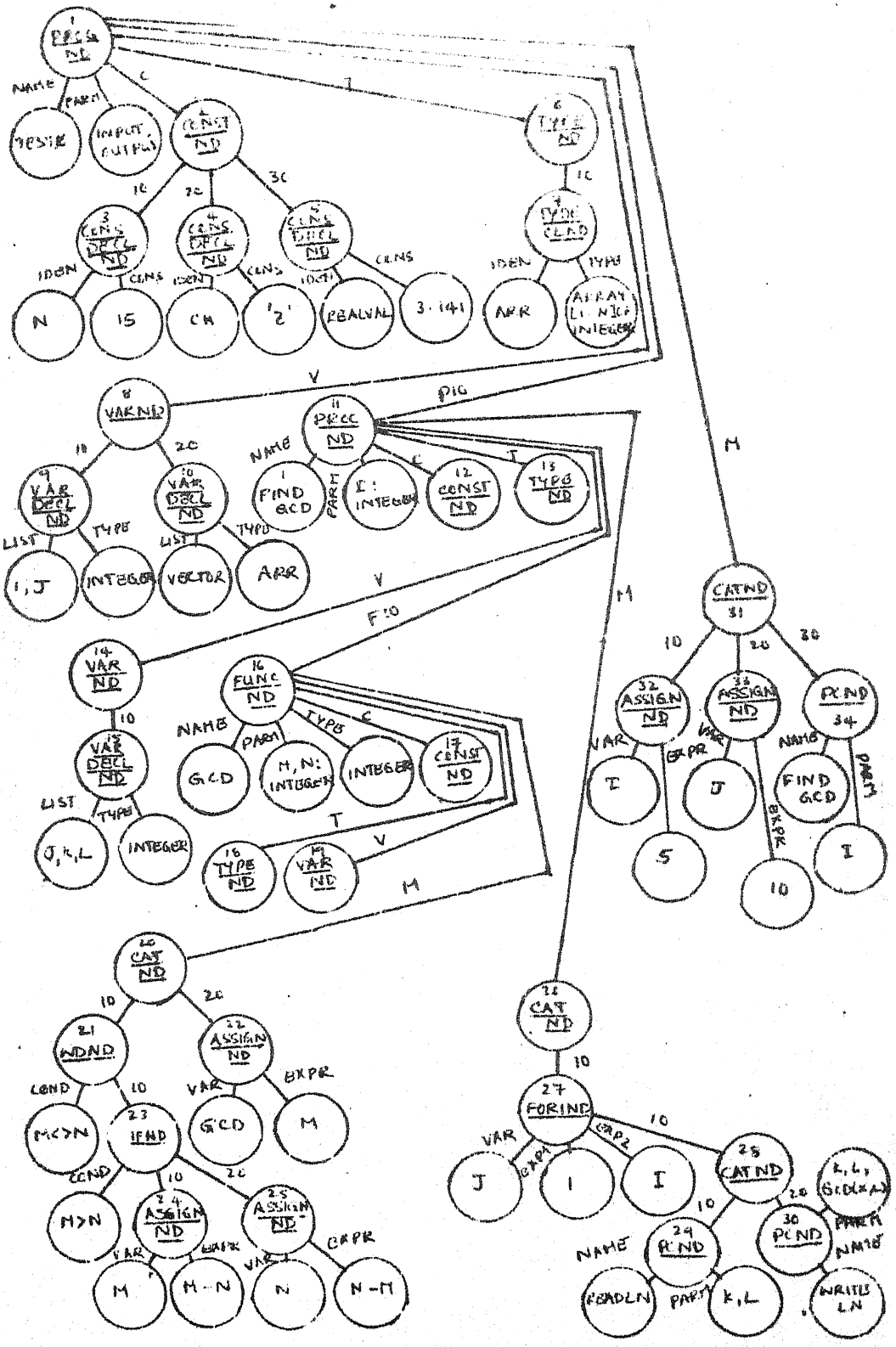


Fig 3.2 Tree Tree For The Example Program in Fig 3.1

accomplished through cursor positioning. The cursor can be moved backwards and forwards in steps through the program and can be used to point at a whole program unit: Template (a predefined, formatted pattern of characters and punctuation marks. The key-words, punctuation and indentation format of a template can't be altered), phrase (an arbitrary sequence of typed symbols) and place holder (identify locations where insertions are permitted). Editing commands allow operations on these entities. Implementing such a scheme requires the availability of intelligent and high speed video terminals.

We have chosen a scheme, in which each node in the program flow tree has a unique number, which can be used to refer to it. A node in the flow-tree has a specified number of sons and each edge from the parent node to the son node is marked by an identification tag which is a sequence of 4 alphanumeric characters called its son number. The node number of any node in the flow-tree is the concatenation of son numbers which occur in the path from the root to the current node. Program constructor can now provide commands like delete, insert, replace or alter a node identified by its node number. In Fig. 3.1, we have given a sample Pascal-S program and in Fig. 3.2 we have given its flow-tree representation. All the non-terminal nodes in this flow-tree are marked by numbers for

```
type
    nodetypes=(nil,progno,constdecl,typedecl,vardecl,procdecl,typeid,
               varid,procid,funcid,pcno,void,proc,assign,call,ifnode,forstmt,
               forupto,casend,unreachable);
    recTypes=(nilrec,nodekrec,linkrec,boundrec,tagrec,startagrec,stantrec,
              arrayrec,procstrec,stackrec,codekrec,flagsrec,fcbrtyprec,cstyprec,
              casestrec,startrec);
  status = (ok,bad,infored);
  {record size 100 / cpu =
        packed record
                        wrk:integer;
                        case recType:recTypes of
                            nodeRec: (nodeId:nodeTypes;nodeLineNumber,prevAddr,compJmpTabAdr;
                                      st:stats);
                            sourceRec :(source:packed array[1..19] of char);
                            linkRec:(link:array [1..3] of integer);
                            subRec:(subno:array [1..4] of char);
                            tagRec: (tagKind:tagClass;tagAddr:integer;
                                     tagType:recType
                                     tagVar:boolean;tagVal:-maxint..maxint;
                                     adr:integer);
                            arrayRec: (indexType,elemType:integer;size:integer);
                            startRec: (dx):integer);
                            stackRec: (value:real);
                            strRec: (nofChars:integer;s:packed array [1..1]
                                     of char;filler:integer);
                            caseRec: (noOfCases:1..maxCases;caseStrs:array[1..3] of
                                     caseRec);
                            flagRec: (fFlag,oFlag:boolean;procLevel:addr);
                            fcbTypRec: (x1:file);
                            cstyRec: (cstyp1:types;cstyp2:integer);
                            tokenRec:(lastToken1,lastToken2:integer);
                            headRec:(nodeRec,tokenRec,exprOp:integer)
                end;
```

Fig. 3.3 Description of the SSPF record definition

use in Chapter 5. From this tree it should be apparent how the scheme works.

3.2 FORMAT AND STORAGE OF SSPF

The nodes in an SSPF may be of various types depending on the programming language. In our programming system we have used a total of 20 possible node types. These have been listed in Appendix A (see Fig. 3.3 also).

A node in an SSPF always contains basic information related to the node; other types of information may or may not be present. Appendix A lists for each node type its compilation related information also.

The nodes in a flow tree are mapped on to a set of records in a random-access file. These records are numbered 1 through n, where n is the size of the file. These numbers can be used to access the corresponding records. There are totally 16 possible record types to accommodate various kinds of information in the SSPF. Complete description of each can be found in the definition of types information (in INCOMP) and cell (in PASED), in the program listings and is also shown in Fig. 3.3.

Each node is associated with a single noderec record and a set of linkrec and sonrec record pairs. linkrec record contains pointers to the sons of a node and sonrec record contains corresponding son numbers. Each of these two records

can accommodate at most 3 sons. Since a node in a flow-tree can have arbitrary number of sons, depending on its type (ex: catnd) to accommodate extra sons, a list of these pairs are used, linked through the field next in linkrec.

A noderec record has the following fields in the specified order. Nodekind, nodenumber, pparent, compinf and st. The field nodekind indicates the node type, Nodenumber holds a number specifying its position in the random access file. The field pparent provides a pointer to its parent node where as compinf points to a list of records representing its compilation - information.

The field st represents the status of the node. It can have three values: new, old or errors. If the value of st is new then the node is said to be a new node. Such a node does not contain any compilation-information. If the value of st is old then it is said to be an old node and is associated with compilation-information. If st is errors then the node may contain some errors. It also does not contain any compilation-information.

Certain nodes represent source text. (only terminal nodes). This is stored in a sequence of sourcerec records linked by the field next. Each sourcerec record contains about sbufflng (= 19 in our programs) characters. The length of sbufflng is to be decided in consideration with

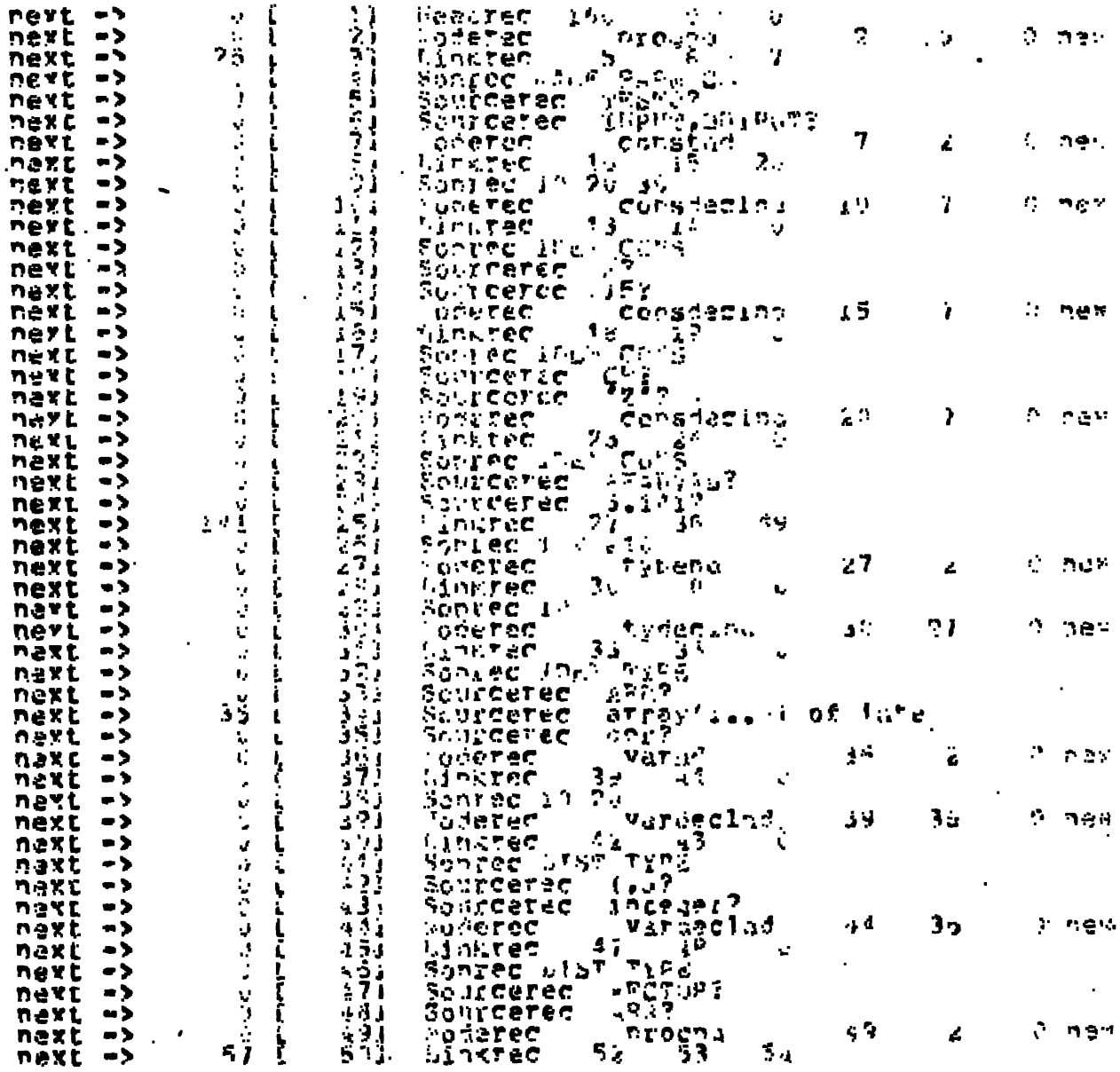


Fig. 3.x SSPF for the example program in Fig. 3.1

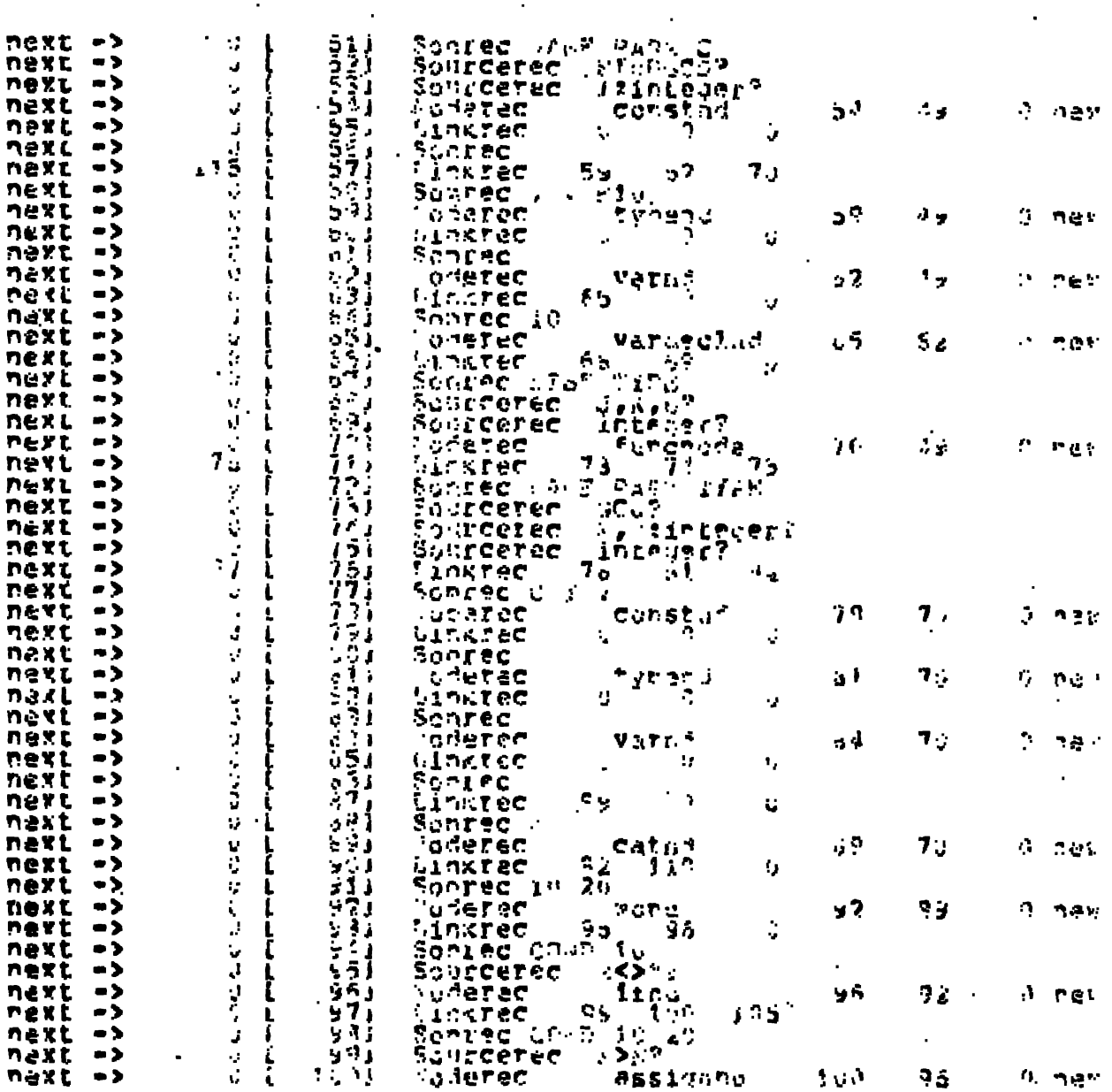


Fig. 3.4 (Continued)

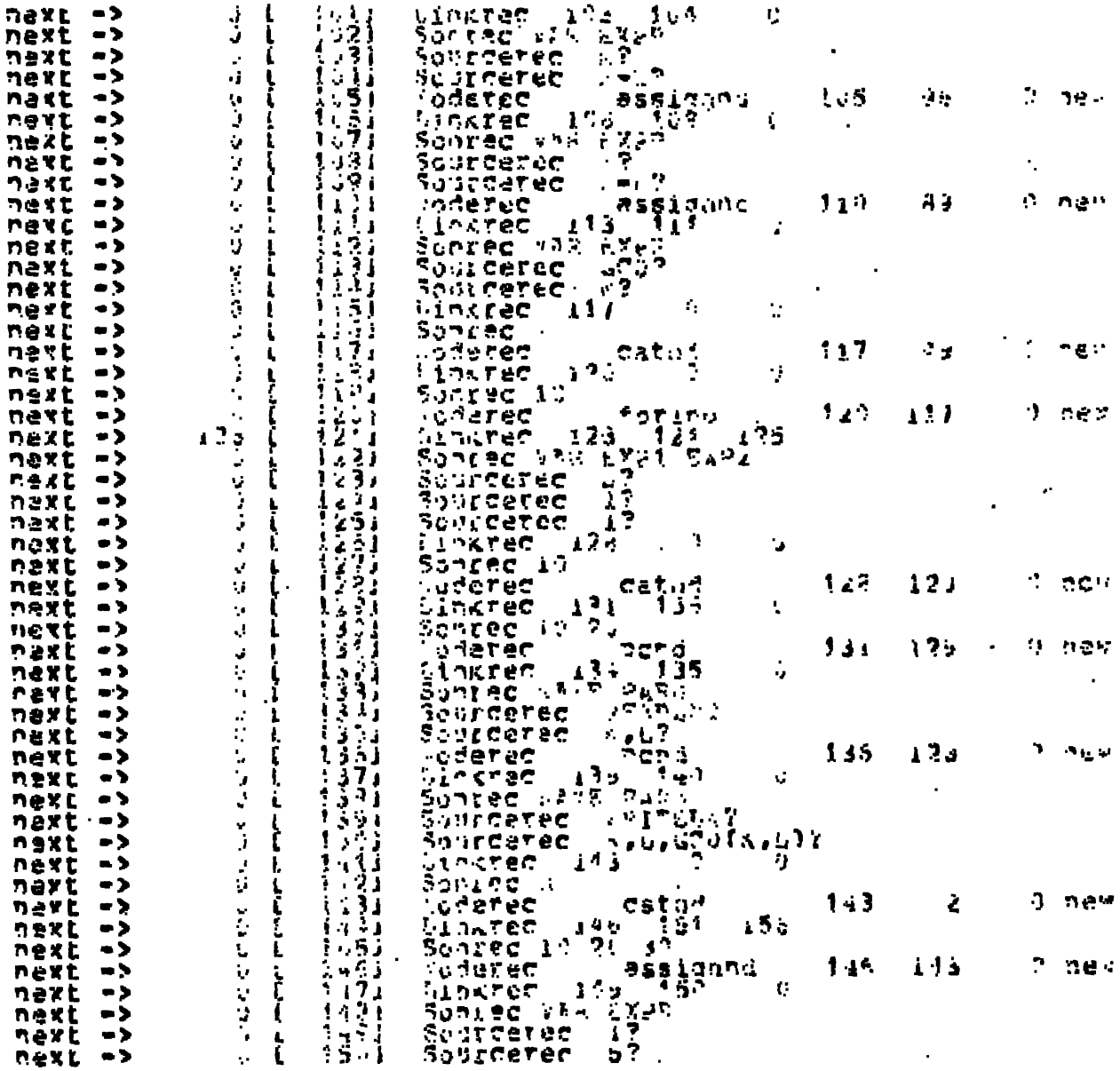


Fig. 1.3 (Continued)

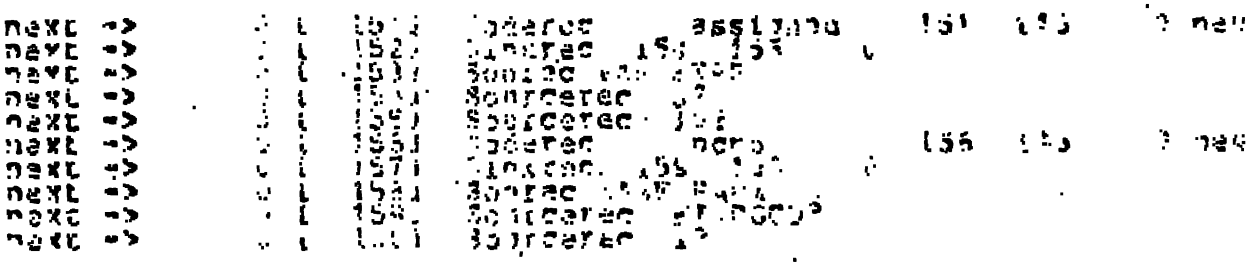


Fig. 5. (Continued)

the size of the other records. The last sourcerec record contains the character '?' (eos) to signify end of source. The description of other nodes is discussed in other chapters.

Fig. 3.4 shows the SSPF created for the program in Fig. 3.1, showing how various records are linked.

CHAPTER 4

PROGRAM CONSTRUCTOR

A program constructor is a key component in any programming system based on structure of source programs. Its basic feature is that it is aware of the structure of the source programs and allows the programmer to view them as structured objects.

In this chapter, we have given an overview of the functions of a program constructor and described the implementation of PASED, a program constructor, we have implemented for the programming language Pascal-S. Our program is based on a syntax oriented editor for Pascal-S, described in [PAR 81] and is essentially the same, with the required modifications. In this chapter, we have used the terms 'Program Constructor' and 'editor' interchangeably.

4.1 OVERVIEW OF A PROGRAM CONSTRUCTOR

A program constructor is basically an editor but differs from the existing editors by providing its users with an environment that permits a structured view of programs. We can formulate several functions it is required to do.

4.1.1 Program Creation

Accepting programs from the user and creating the

corresponding flow trees and preserving them in syntactically correct form is the program entry function. This involves accepting the source text and performing the necessary steps to map it into a flow-tree form (this may involve parsing) and ensuring the absence of block structure errors (without which formation of flow tree is impossible). Some form of syntax prompting can be provided to aid the user.

4.1.2 Program Traversal

Before performing editing one needs the capability to selectively display and traverse parts of the program. The program constructor must allow the user to position himself at various points in the program and delineate a certain portion around that point. Since we are storing our programs as a tree, this consists of being able to identify a node in the flow tree and to refer to the subtree rooted at this node. We can provide position dependent traversal in which user can refer to the parent or son of the current node and position independent traversal in which user can refer to any arbitrary node in the flow tree.

4.1.3 Program Editing

Deleting program segments is the easiest task the editor can be asked to do. This involves clipping sub-trees representing the offending portions. Thus entire procedures,

statements, declarations, etc., can be deleted. However, some parts like the condition part of a while - do statement cannot be deleted. The editor has to guard against operations which may introduce such errors.

Insertion of a new section is very essential. This involves constructing a flow-tree segment for this section and attaching the root of this newly created segment as a son to the appropriate node.

The delete-insert operations can be combined to give a Replace operation which can replace individual nodes of any kind or whole flow tree segments.

Apart from these basic operations, the editor can be equipped to transfer, copy and alter various nodes or flow tree portions. It can also have operations that search for a node containing a given string or substitute a string for another string in a flow tree segment.

4.2 PROGRAM DOCUMENTATION OF PASED

The format of the SSPF for Pascal-S programs is outlined in Chapter 3. This format is essentially reflected in the definition of the record type cell in the PASED program (see Fig. 3.3).

Nodetypes and Rectypes represent the types of nodes and records present in the SSPF respectively. PASED makes use of only 4 kinds of records : Headrec, Nodrec, Linkrec, and

Sonrec. The rest of the records are used in the incremental compiler INCOMP (Chapter 5).

Some variables of importance are given below :

Stat is a variable that fetches a statement type from the user. Keytab is an array of key word strings used in changing the key words into lower-case strings. Err is a variable that is set to an appropriate value, whenever, an error occurs and is subsequently used in errout procedure to display an error message. Pat and pat1 are two string buffers used in Found and Substitute procedures. Buf1 is a buffer for holding source text of a particular node, used in alter it, alter, and substitute procedures.

Following are some variables local to the procedure edit : Curline is an array of records, each record having a son number and an address of the node. It stores the current node pointer. It is used as a stack with curlev as a stack pointer.

Newprog is initialized to true, when entering a program for the first time and is used to skip changestatus.

4.2.1 Random Access File Routines

The programs are written in Pascal. Pascal on DEC-10 does not support random I/O (input/output). But our programming system requires such random access. Hence, a random access file package has been implemented in MACRO-10

[PAR 81]. It has three procedures.

```
procedure  select (var F: filetype);
procedure getrecord (var F:filetype; i:integer);
procedure putrecord (var F: filetype; i:integer);
```

which have to be declared as external procedures in any program using them.

Select initializes a file for random access I/O and getrecord and putrecord read and write respectively at the specified record position.

4.2.2 Procedures for Program Entry

Procedures getnode and putnode get and put a record respectively in the file tmp using the external random-access file routines. The procedures Readbig and Readsmall are external procedures for reading a character from tty. They both return a CR LF combination, only as LF . Readsmall reads characters as it is and Readbig converts them to upper case before reading. Readtty reads a character from tty using these procedures and the value of quote.

Inchrw, echo and noecho are also external procedures. Inchrw also reads characters from tty and makes them available as soon as they are typed, without pressing RETURN . Noecho suppresses echoing of characters typed and echo enables this. They are used in alter.

The main program gets a filename using getfilnam. If the file is not already there on the disk then it enters the program entry mode by calling proc with level = 0.

Proc and func are procedures to enter a procedure and a function respectively. As a program is a special case of a procedure, it enters a program if level = 0. The variable level denotes nesting a depth of the current program segment in the flow-tree and is used for indentation and printing of node numbers. They have arguments nodenum, lnodenum and snodenum which represent node numbers where the flow-tree segment for the procedure/function to be entered is stored. They are used for filling the pointer information.

Proc enters the name and parameters of a procedure and then calls block to enter constant declaration, type declaration and var declaration by calling the appropriate procedures (suggested by their names) and then calls proc/func to enter inner procedures or functions.

Finally block calls csst to enter a compound statement. Procedure statement enters a statement. It uses gettype to get a statement type and calls the appropriate procedure. The procedures to enter various statements should be readily apparent if gone through, as they are written, strictly according to Pascal-S syntax.

All declarations, variables and expressions are accepted from tty at the request of the accepting routines through procedure readexpr. It reads a string from tty converts identifiers to upper case, key words to lower case and leaves literals alone. It generally uses RETURN as an end of source string unless when entering a Record in which case, it uses it to indent fields and keeps accepting strings until end is encountered.

4.2.3 Procedures for Program Modification

Edit is the main procedure containing all the procedures needed for editing an SSPF. The main program calls edit when a file that already exists is given a after entering a new program.

The statement part of edit consists of a loop which gets a command (getcomand) and does it (docomand) if there are no errors. If any error occurs, during any of the editing procedures, variable err is accordingly initialized and an appropriate messages will be displayed.

The significance of procedure changestatus is discussed in Chapter 5. This procedure is not required for any new program.

Getcomand gets a command from the tty in com and any arguments like nodenumber, sonnumber, registernumber etc. depending on the command. Getentry gets a sonnumber from tty

and concatenates it with the parent nodenumber to get the nodenumber of the son in gotoline. Getlineno gets a new node number.

Procedure delete identifies the son to be deleted of the current node and deletes simply the pointer in the parent node to that node, if the syntax permits it.

Procedure insert inserts a pointer at an appropriate place to the new flow-tree segment created for the inserted section.

These two procedures use savelink and Restorelink. Savelink saves all the pointers of a node to its sons, which occur to the right of the given son. Restorelink restores these to the node. They are used to shift the pointers in case of a delete or an insert operation.

Procedure Found searches the source nodes in the given sub-tree for an occurrence of the pattern in variable pat. and displays that node along with its node number. It uses function pmatch for this. pmatch is a pattern matching routine that is based on the Knuth-Morris-Pratt linear algorithm . Its value is true if a pattern match occurs and pos will be the position in the source marking the beginning of the pattern; otherwise it is false.

Similarly, substitute replaces all the occurrences of the string in pat by the string in pat1 in the given sub-tree,

It also uses pmatch for pattern matching.

Alterit is the procedure that alters a given node,. It it is a source node, it is copied into a buffer, buf1 and alter is called. Alter permits modification of buf1 and it is copied back into the original source node at the end. If the root of a sub-tree is given to alterit, it alters all the source nodes in that sub-tree.

Displayersnodes displays the list of nodenumbers which represent erroneous nodes. It takes an argument, a pointer to the 1st record in a list of linkrec records and follows each link of this record which again points to a list of sonrec records to display the nodenumbers.

Updateerrors, checks this list of nodenumbers after the end of each command that results in the modification of SSPF and if currently modified node number occurs in this list, it is deleted.

The procedures which transfer and copy nodes are very straight forward as they use delete and insert operations as primitives.

Saywhat displays the nodenumber of the current node and all its son numbers if there are any.

Display takes as argument, a nodenumber level and depth and displays the sub-tree rooted at this node on the tty as a user readable text to the specified depth. It uses a

Compress is the procedure which deletes all the garbage accumulated during editing and compacts the file. It starts with root node and traverses the program tree with all the compilation information and copies it into file tmp2 and in the process discards all those records which do not occur in the program tree. After compaction, file tmp2 is copied back into main.

Our description of program PASED has been very sketchy and more detailed information can be obtained by going through the program listing.

A list of the procedures, which occurs in the program PASED, showing their static nesting is given in Appendix B.

CHAPTER 5

INCREMENTAL COMPILER

Structured Source Program Files created by the Program Constructor are amenable to incremental compilation. That is, after a program is modified only the modified part need be recompiled. This is a desirable feature particularly for programs under development.

In this chapter, we have described briefly the design/ implementation of an incremental compiler which works on the SSPF created by the program constructor PASED. We have started with a version of the compiler described in [JAJ 80]. This compiler has been implemented, just to prove that incremental compilations of programs in a block structured language is possible. It works on a sequential file on to which the Pascal-S flow tree is mapped. Since Pascal on DEC-10, the language in which this compiler was originally written, allows only sequential files, the nodes in the flow tree are arranged in the file in a specific order known to the compiler. On such a file, the editing environment we have discussed in Chapter 1 cannot be implemented. Hence, we have implemented another version, which can now work directly on the SSPF created by the program

To properly understand the design of a compiler it is necessary to know both the features of the source language

to be compiled and the architecture of the underlying machine on which the generated code is to be executed.

A description of the main features of Pascal-S, the source language can be found in Chapter 6. The compiler produces intermediate code and if compilation is successful, it interprets the code generated. Section 5.1 is a description of the machine on which the object code is interpreted. Section 5.2 presents an overall view of the compiler. The symbol table and other tables used in the compiler are discussed in Section 5.3. The crux of this chapter is on how incremental compilation is achieved and can be found in Section 5.4. Section 5.5 describes the set of nodes to be recompiled after a program is modified. The implementation aspects are discussed in Section 5.6 and Section 5.7. Concludes this chapter by outlining the method used for debugging the programs. Sections on Pascal-S machine and symbol tables and other tables have been taken from [JAJ 80], and are given here for the sake completeness. These descreptions can also be found in [WIR 75] also.

5.1 PASCAL-S MACHINE (INTERPRETER)

The compiler produces code for a hypothetical stack machine (which we call Pascal-S Machine). This machine is itself defined as an algorithm the interpreter of the compiled code. We have used the same interpreter as the one used in [WIR 75]. This is encompassed in procedure interpret in the

program. For the sake of completeness we have discussed briefly its organization.

The interpreter has a small store S, organized as a stack. Two index registers T and B control the stack. A program counter PC, an instruction register IR, a program status register, PS and a display to speed up access to non-local variables, are also there. Each element in the stack is either an integer, a real number, a boolean value of a character. The main interface between the compiler and the interpreter is the array code which holds instructions generated by the compiler. The primary structure of the interpreter is :

```
procedure interpret;
   begin  initialize all the registers and auxiliary
              counters;
      repeat
         ir:= code [PC]; PC:= PC+1;
              interpret ir
      until PS ≠ Run;
      if PS ≠ fin then postmortem dump
end;
```

Each instruction has 3 fields : f, x and y. f is an operation code (opcode) and can take a value between 0 and 63. Opcodes 0,1,2 generate an address of the data element, with offset y in the currently active data segment on level x

and loads it on top of the stack. Opcode 3 updates display by using fields x and y, opcode between 8 and 30 need only field y. Opcodes 31 to 63 need neither of the fields x and y. Their arguments are assumed to be available on the top of the stack.

5.1.1 Procedure Calls and Storage Layout

On a procedure call, a stack section (activation record) is reserved for the called procedure on the top of the stack. The beginning of this stack section is a special mark called sectionmark. The sectionmark contains a pointer called the dynamiclink to the beginning of the stack section corresponding to the calling procedure. The chain of dynamic links traces out the dynamic history of the program execution.

The sectionmark also contains a staticlink which is a pointer to the stack section corresponding to the procedure in which the currently active procedure is declared. The chain of static links specifies the static nesting of the procedures and is necessary in determining all the accessible data names for the currently active procedure. This chain is also copied in display to speed up the accessing.

Apart from these two, a sectionmark also contains the returnaddress, a pointer to the symbol table and a location to store the result value (used only for functions).

All the parameters of a procedure are processed and storage is allocated for them on the top of the stack,

immediately above the section mark. The storage for local variables is allocated after this. In case of value parameters, only their values are loaded and for variable parameters their addresses are loaded. All the variables are initialized automatically. Finally, instruction registers B and I are updated properly and PC is set to point to the first instruction of the activated procedure.

Upon exit from a procedure a return instruction is executed, which reverses the operations performed by a call instruction. If the static level of the current procedure is lower than that of the procedure to which control returns, display is properly updated.

Opcodes 3,18,19,32 and 33 perform these operations.

5.2 OVERVIEW OF THE INCREMENTAL COMPILER

The SSPF created by the program constructor forms the input to the incremental compiler which produces the object code for the interpreter. A large part of the compilation information (such as symbol table entries) generated during compilation, which is used subsequently to derive the object code is preserved in the SSPF in a suitable form, so that it can be reused in subsequent compilations. The central idea is to associate a packet of compilation information to each node in the SSPF. This incremented SSPF is available for modification to the editor to reflect the changes desired by the programmer. Recompilation of the modified SSPF then

requires deriving information only for those nodes, which are new and linking it together with information associated with other nodes.

The compiler expects its input in a (logical) file called iprogf. It is associated with a physical file at runtime. The file is then copied into a temporary file called tmp (this is necessary in view of the random access file package used), and works on this temporary file. After compilation is over, the incremented temporary file is copied back into iprogf.

The tasks performed by the incremental compiler can be broadly divided into :

1. to get and test records in file tmp
2. to scan the source records associated with a new node
3. to copy compilation information already available in tmp
4. to construct records for newly generated compilation information and to append these to the corresponding nodes for use in subsequent compilation
5. to handle symbol table and other tables
6. to analyze individual nodetypes.

The compiler contains a collection of procedures and is discussed further in Section 5.6.

For proper understanding of the compiler, we can divide it into 2 parts :

1. The part processing declaration nodes : nodes of type prog, proc, func, const, type, var, consdecl, tydecl and vardecl
2. The part processing statements and expressions : nodes of the type cat, if, wd, ru, for1, for2, case, onecase, and assign.

Their common interface is symbol table tab and its associated tables. These tables are constructed, while processing declaration nodes and are then used to derive the context in which other nodes are compiled. Knowledge of these tables is therefore very essential.

5.3 COMPILER TABLES AND THEIR STRUCTURE

The key table is tab. The index variable for this table is t, and always points to the last entry made in tab. Each declared identifier results in one entry. All identifiers belonging to the same procedure (block) are linked together using the field link. The link field of the first identifier declared is set to 0; The entry with index 0 is used as a sentinel for searches on tab.

The field obj specifies whether the declared identifier is a constant, a variable, a type, a procedure or a function. The type of the identifier is indicated by the field called typ. The meaning of the remaining fields is dependent on the contents of these two fields.

If the object denoted by the identifier is a procedure/ function or a type of records then, the ref field is an index to the block table btab. If the identifier is of type arrays then ref is an index to the table of arrays, atab. In all other cases it is 0.

The field normal is meaningful only for identifiers, which denote variables. It specifies whether an identifier is an actual variable or a value parameter to be addressed directly or a variable parameter to be addressed indirectly.

The procedure block uses the variable dx to allocate storage on the stack section. The value of this variable is used to set the adr field of identifiers that are variables. The field lev records the level at which the identifier is declared. The pair lev, adr thus forms an address for any variable. If any entry is a procedure or a function the adr field points to the first instruction in this procedure or function. If the entry is a type identifier then the value of adr field specifies the size of the variables of this type. If the entry is a constant, then if it denotes a real constant, adr is a pointer to a table of real constants called rconst. For any other type of constant, adr field directly specifies the value of the constant.

The table atab stores the information regarding every declared array. The fields inxtyp, low and high specify

indextype, and indexbounds. The fields eltyp and elsize indicate element type and element size. The field elref is used as the field ref in tab. Although, elsize can be obtained from eltyp and elref, it is stored for convenience. The total store needed for the array is given by size.

Each procedure, each record definition causes an entry in the block table btab. The fields lastpar and psize are used only for procedures. They specify the last parameter of the procedure (by the use of index to tab) and the total store needed for the parameters. The field last indicates the last identifier declared in that procedure. The value of the field last is updated as each new entry is made. The field vsize specifies the total store needed for the variables declared in that procedure.

In addition to the fields described above the 3 tables tab, atab and btab have a special field called token. This field is added for providing a unique identification of individual entries of the tables. Their use is further explained in later sections.

For the above table, the entries created during the compilation of a procedure can be disposed off when the compilation of that procedure is over. However, they are retained as they are needed in performing postmortem dump, in case of any run time errors. This makes it unnecessary to copy certain information into the code, for example, data

Fig. 5.1 Various compiler codes produced for the example program in Fig. 3.

segment length and array index bounds; an instruction requiring such information contains an index of the relevant entry.

Moreover, due to the chosen nature of the interpreter, the compiler has to create and maintain two more tables viz., rconst and stab, for their use in the interpreter. The table rconst contains all the real constants used in the program. Since the instruction format does not permit passing a real constant, in the code, the index of the relevant entry is used in its place.

The table stab contains strings that are used as parameters of procedure write. In the code generated, for a call to this procedure, a reference may be made to this table to indicate the desired string.

We have given an example, in Fig. 5.1, showing the tables tab, atab and btab and code generated for the example program in Fig. 3.1.

5.4 INCREMENTAL COMPILATION

Compilation of an SSPF involves compiling of every node in it. The compilation of a node produces a) the object code corresponding to this node and/or b) the information necessary to update the global compiler tables. Any such information is referred to as compilation-information.

The principle of incremental compilation consists of preserving as far as possible the compilation information produced in one compilation of a program, so that it can be used in subsequent compilation, by avoiding duplication of effort. This may be done by storing with each node, the compilation information it produces. If the source contained in this code, remains unaltered, then in a subsequent compilation, it need not be compiled. In such case, the preserved compilation-information is simply copied.

There are however, cases when recompilation of a node becomes necessary due to changes in other nodes (or addition of a new node), even though the source code in that node is not altered. To cater to these cases, we can derive rules, to identify all such nodes and this, we discuss in the following section.

Compilation-information usually contains a reference to the compiler tables, Such a reference is usually specified in terms of the corresponding index of the referred entry. However, physical position of an entry in the table may vary from compilation to compilation. Hence, we associate with each entry a unique identity token (which is simply an integer) that remains invariant from one compilation to another and we use this instead of the corresponding table index. When we are copying the compilation-information, we can convert these tokens

to table indices by searching the token values in the appropriate table.

In our implementation, identity tokens are created for entries in table, tab, atab and btab. Entries in rconst and stab are copied whenever there are references to them. Since entries in stab are referred from only one place there is no duplication, which may not be true for entries of rconst.

5.4.1 Compilation Information of Different Node Types

Appendix A gives the compilation-information which is stored with nodes of various types. This information is stored as a list of records, with a pointer to this list in the corresponding noderec record in the field compinf. The possible types of records used for this purpose are tabrec, atabrec, btabrec, bta1brec, rconstrec, stabrec, coderec, flagrec, exprtyprec, cltyprec and tokenrec.

Nodes of type cat, if, wd, ru, for1, for2, case, onecase, and assign do not produce any contextual information. We may consider only, what they contribute to object code.

Catnd contributes no instructions to the object code. So no compilation-information need be associated with this. Though, if, wd, ru, for1, for2 and onecase contribute only jump instructions, they contain compilation information generated from their terminal sons (source nodes). For

example, if, wd, ru nodes have a boolean expression as a son, for1 and for2 nodes have a variable and two expressions as sons and so on. Onecase node contributes apart from jump instructions, only those instructions, which can be derived from the compilation-information, stored, with the associated clabel nodes.

Case node contains compilation-information about the case expression. Expressions may use real constants and hence may contribute instructions which make references to rconst table. Because of the way in which the rconst table is constructed (to avoid duplicate entries), it is possible that some of the entries that are referenced may have been caused due to some other nodes. If any of these nodes gets recompiled due to some reason, then the entry referenced may not exist during next compilation. To overcome this, we directly store the real constants referenced by the contributed instructions in rconstrec records.

Assign nodes and expression nodes may also make references to tab and atab tables. Therefore, before storing these instructions, the references have to be converted to corresponding identity tokens.

Since a procedure call node (PC) may be a call to the procedure write, it may require a part of the stab table, to be copied in the compilation-informations. It is for this purpose, stabrec records are used. Each stabrec can

accommodate upto 18 characters. If the string length is more than 18 then a sequence of stabrec records is used linked by the field following.

Some nodes require the type of their expressions sons to be included in the compilation-information. This is used to check type compatibility. So exprtyprec records are used for this.

For case label nodes, we need only store the list of constants and their type.

The flags which indicate whether input, output are declared in the program heading or not have to be stored in the compilation information for prog node. So flagrec record is used for this purpose. Along with these flags, the program name, which is used in the compiler to produce a listing of compiler tables in the file output, is also stored.

The nodes to be discussed below require storing entries of tab, atab, btab tables. The link and ref fields (when the latter is not 0) of the table tab and the field elref of the table atab contain indices. While storing ref and elref these are converted to corresponding tokens. It is better to regenerate link whenever required than to use tokens . Since the fields last and lastpar are dynamically updated, they need not be stored.

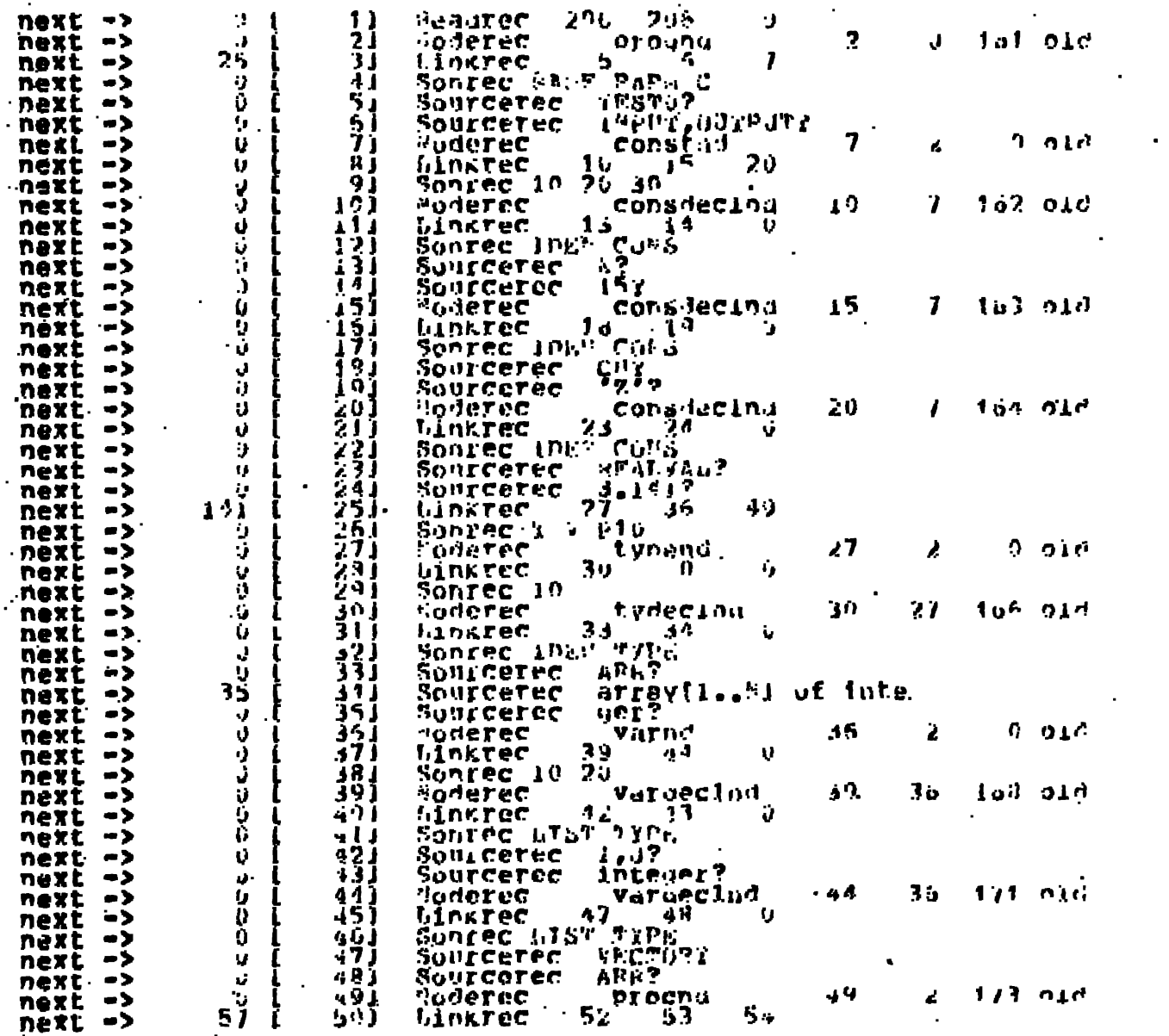


Fig. 5.2 Incremented version of the SSDF in fig. 3.4

```
next ->       0 L   51] Sourrec , ASF PARM= C
next ->       0 L   52] Sourcerec  FUNCTION
next ->       0 L   53] Sourcerec  integer?
next ->       0 L   54] Coderec      constod          54   49     0 010
next ->       0 L   55] Linkrec      0    0    0
next ->       0 L   56] Sonrec
next ->     115 L   57] Linkrec     59   62   70
next ->       0 L   58] Sonrec I = 1,N
next ->       0 L   59] Coderec      typeno           59   44     0 010
next ->       0 L   60] Linkrec      0    0    0
next ->       0 L   61] Sonrec
next ->       0 L   62] Coderec      varod            62   49     0 010
next ->       0 L   63] Linkrec     65    0    0
next ->       0 L   64] Sonrec 10
next ->       0 L   65] Coderec      varuclud         65   62   176 010
next ->       0 L   66] Linkrec     68   69    0
next ->       0 L   67] Sonrec ATSV TYPE
next ->       0 L   68] Sourcerec  a,n,b'
next ->       0 L   69] Sourcerec  integer?
next ->       0 L   70] Coderec      funcnode         70   19   160 010
next ->      76 L   71] Linkrec     73   74   75
next ->       0 L   72] Sonrec ASSF PARM TYPE
next ->       0 L   73] Sourcerec  aCO?
next ->       0 L   74] Sourcerec  a,Aainteger?
next ->       0 L   75] Sourcerec  integer?
next ->      97 L   76] Linkrec     78   81   84
next ->       0 L   77] Sonrec C 1 1
next ->       0 L   78] Coderec      constod          78   79     0 010
next ->       0 L   79] Linkrec      0    0    0
next ->       0 L   80] Sonrec
next ->       0 L   81] Coderec      typeno           81   79     0 010
next ->       0 L   82] Linkrec      0    0    0
next ->       0 L   83] Sonrec
next ->       0 L   84] Coderec      varno            84   79     0 010
next ->       0 L   85] Linkrec      0    0    0
next ->       0 L   86] Sonrec
next ->       0 L   87] Linkrec     89    0    0
next ->       0 L   88] Sonrec A
next ->       0 L   89] Coderec      catod            89   79     0 010
next ->       0 L   90] Linkrec     92   10    0
next ->       0 L   91] Sonrec 10 20
next ->       0 L   92] Coderec      vano             92   99   101 010
next ->       0 L   93] Linkrec     95   96    0
next ->       0 L   94] Sonrec LOGD 10
next ->       0 L   95] Sourcerec  A<>B?
next ->       0 L   95] Coderec      lino             96   92   165 010
next ->       0 L   97] Linkrec     99  100  105
next ->       0 L   98] Sonrec COMP 10 20
next ->       0 L   99] Sourcerec  A>B?
next ->       0 L  100] Coderec      assigno         100   96   146 010
```

Fig. 5.2 (Continued)

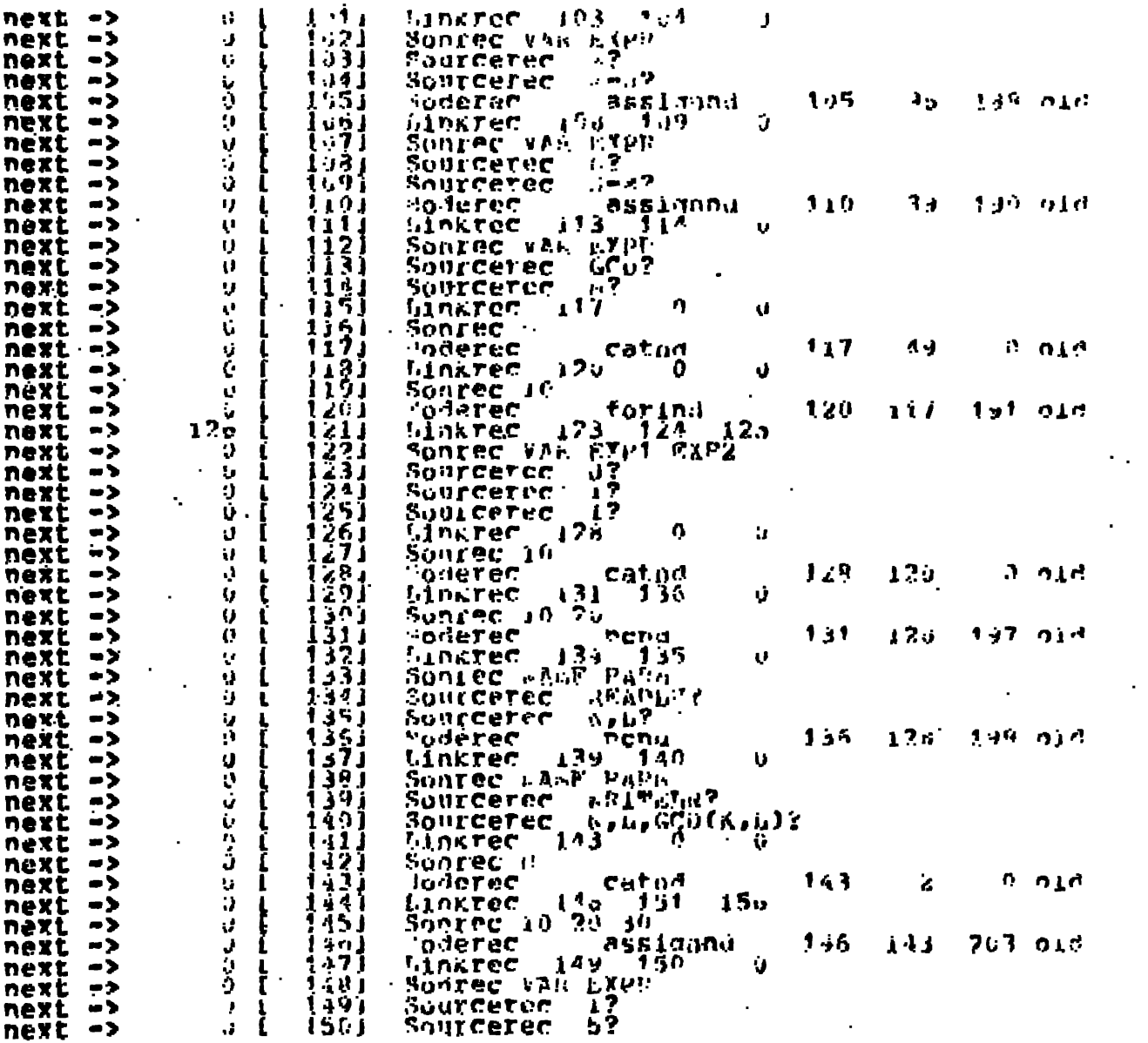


Fig. 5.4 (Continued)

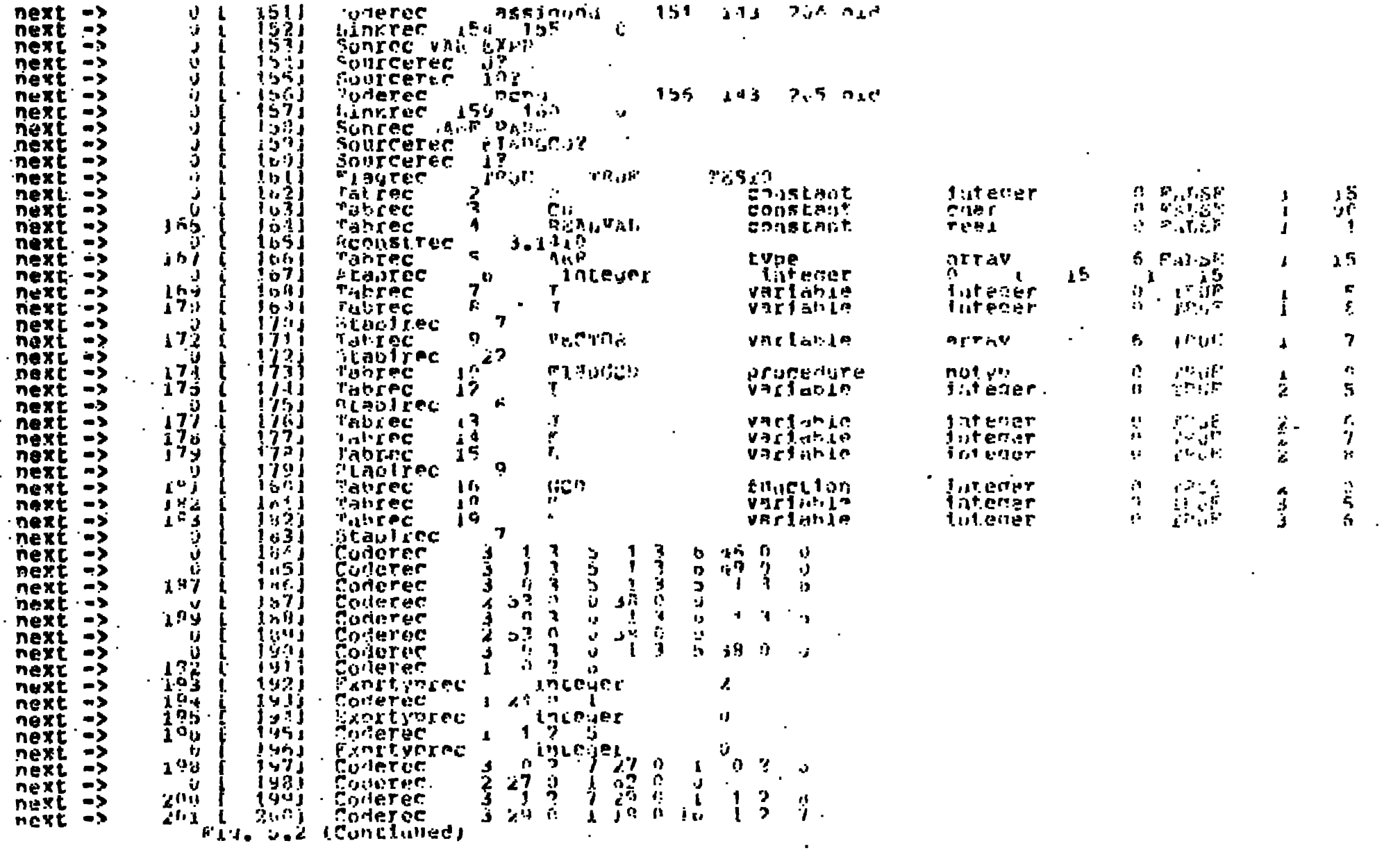


Fig. 5.2 (Continued)

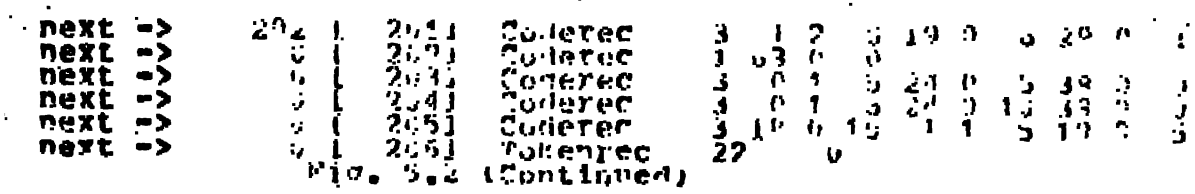


Fig. 5.2 (Continued)

Each proc/func node causes an entry in the table tab. The adr field of this entry points to the first instruction of this procedure/function and is to be derived anew for each compilation.

Proc/func nodes also contain certain tab records for their parameters. Also, the value of the variable dx which indicates the size of the store needed to load the parameters is also preserved in btab1rec record.

Const, type, and var nodes do not generate any compilation-information.

Consdecl node contains tab and/or rconst entries. However, the ref field in tab need not be converted to identity tokens, as its value will be zero for constants.

Tydecl and vardecl nodes can cause entries in tables tab, btab and atab and hence these are included in these nodes in corresponding records. In btab entries, only psize and token fields need to be preserved. For a vardecl node, we also store dx.

Fig. 5.2 shows the incremented version of the SSPF in Fig. 3.4 and shows how the compilation-information is stored.

5.5 SCOPE OF RECOMPILATION NECESSITATED BY CHANGES

If an already compiled program (SSPF) is modified by the program constructor, obviously those nodes, whose source has been modified must be recompiled. Hence, their status has to

TABLE 3.1

X	NODE UNDER SOURCE IS ALTERED	Y	PARENT NODE OF X
Z	PARENT NODE OF Y	S	SET OF THE NODES WHICH HAVE TO BE RECOMPUTED

MODIFIER	SOURCETYPE	SET OF NODES TO BE RECOMPUTED	SYNTAX (REFER TO FIG. 3.2)
PROGID	PROGRAM NAME, PARAMETERS	ALL THE ASSIGN NODES IN THE PROGRAM (PEP	X = 1 S = (22,24,25,32,33)
PROCND	PROCEDURE NAME, PARAMETERS	ALL THE DESCENDANTS OF NODE X ALL THE STATEMENT NODES IN ALL THE SUBTREES OF Y WHICH OCCUR AT THE RIGHT OF X	X = 11 ; Y = 1 S = (11,12,13,14 ...)
FUNCND	FUNCTION NAME PARAMETERS	SAME AS ABOVE	X = 16 ; Y = 11 ; Z = 1 S = (16,17,18 ... 30)
CONSTND	NONE	ALL THE SUBTREES OF Z TO THE RIGHT OF Y, WHOSE SOURCES POINT AT X	X = 4 ; Y = 1 S = (2,3,4, ... ,31)
TYPEND	NONE	SAME AS ABOVE	X = 5 ; Y = 1 S = (6,7,8,9 ... 30)
VARND	NONE	SAME AS ABOVE	X = 8 ; Y = 1 S = (8,9,10 ... 29)
CONSUSECOND	IDENT,CONS	ALL THE SUBTREES OF Y WHICH OCCUR TO THE RIGHT OF X AND ALL THE SUBTREES OF Z WHICH OCCUR TO THE RIGHT OF Y	X = 5 ; Y = 2 ; Z = 1 S = (2,3,6,7, ... ,34)
TYDECLND	IDENT,TYPE	SAME AS ABOVE	X = 7 ; Y = 6 ; Z = 1 S = (5,7,8, ... ,34)
VARDECLND	DISP,TYPE	SAME AS ABOVE	X = 10 ; Y = 9 ; Z = 1 S = (9,10,11,12, ... ,34)
OTHERS	NONE		NULL

be made new. But depending on the type of the node modified, other nodes may also be required to be recompiled.

For various node types, the set of nodes to be recompiled is given in Table 5.1. This table is for the case when the node in question is replaced or altered.

Suppose a node has been inserted into the SSPF. This can be treated as equivalent to the above case. We can think that this node is already present in the SSPF, with null source, to which source has been added. Similarly, deleting a node can be taken care of.

The program constructor provides many more operations like transferring which is equivalent to deleting a node from one place and inserting at some other place, copying which is equivalent to inserting a node and substituting which is equivalent to replacing/altering an already existing node.

5.6 IMPLEMENTATION NOTES

The incremental compiler INCOMP works on the SSPF produced by the PASED (program constructor). The organization of SSPF has already been explained.

As has been pointed out, each node (identified by a noderec record) can have as its sons either source code (terminal nodes) or nodes of other types (non-terminal nodes). Each source node occurs as a list of sourcerec records linked by next.

The procedure getfilnam gets a filename and switches if any from the tty. A modified reset command is used to associate this filename with the variable iprogf. The switches are used to initialize option1 and option2. The compiler copies iprogf into a temporary file tmp and works on this. After compilation is over, it is copied back into iprogf which then represents the incremented versions of the original file.

The file output produced by the compiler contains all the node numbers whose status is new and hence compiled, along with their source code and error messages if any. Switches are available to suppress this listing (option2). Output also contains the results of execution.

In the present implementation, each entry of the tables tab, atab and btab is associated with a unique token which remains invariant from one compilation to another. These are simply integers starting with one. To make sure that the generated tokens are indeed unique it is necessary to keep track of the tokens used in previous compilation. So the last token used is to be preserved. This is done using tokenrec record. The Headrec record which is the first record in the file contains a pointer to tokenrec record in the field tokrecno.

Procedures getnode/putnode in the INCOMP programs get/put a record given a location in file tmp. They make use of the

The first record in the file is always headrec record. It contains the number of records in the file and is used to initialize nextnode, which always points to the last record in the file. Using tokrecno, tokenrec record is read and lasttoken1 is initialized. After all the initializations, the main program calls prognode.

The variable curline is an array of records each record containing two fields lineno and nodeaddr. It is used as a stack with curlev as a stack pointer. The pair, curlev, curline identifies at any time the node number and address of the node being compiled. Setline is used to update curline before compiling a new node.

The compilation proceeds by compiling each node in the SSPF. For each node type, there is a procedure to compile nodes of that type. The names of the procedures fairly suggest, which type of nodes they are compiling.

In general, each procedure does the following :

The status of the node is first checked. If its value is new then the node to be compiled is new and it has no compilation-information. Therefore, all its source sons are to be scanned.

The scanning of a source son proceeds as follows : Each source son is a list of sourcerec records. The variable currecno is initialized so that it points to the first record

in this list. Then initnextch is called to provide the necessary initializations. After this procedure insymbol is called repeatedly until the entire source is scanned. This is signified by encounting an eos = '?' symbol.

Procedure insymbol uses nextch to supply the next character in the source, via ch. Whenever, CC = esbufflng, Nextch, fills the variable line with following sourcerec record if any and sets CC to 0. It also writes the source in the output file if option2 is true.

After scanning all the source sons, the compilation information generated is stored in the node itself as a list of records, with a pointer to the first record in this list in the field compinf, for use in subsequent compilations. This information is different as discussed for each node (see Appendix A).

If the status of the node is old then it has compilation-information associated with it and is copied by accessing this through compinf. After this, the source sons are skipped.

The non-terminal sons are again compiled by calling appropriate procedure.

The procedure describing the compilation of each node is straight-forward and can be easily understood.

However, the points to be noted are that for compiling

expressions, the syntex of Pascal-S expressions is straight-away used. This results in a set of mutually recursive procedures.

For copying compilation-information, and generating records for storing this, the procedures used are described below.

Outtabrec, outatabrec, outbtabrec create records corresponding to entries in tab, atab and btab tables for storing compilation-information. Any references to other tables are converted into identity tokens.

Outcoderecs is used for storing object code generated for a particular node. Any reference to tab, atab, entries are stored as corresponding tokens. Any real constants a or strings used by these instructions are stored in tables temprconst and tempstab temporarily and are used in generating rconstrec and stabrec records in procedures outrconstrecs and outstabrecs.

Copytabrec, copyatabrec and copybtabrec are used for copying tab, atab and btab table entries. Any references to other tables are resolved using setelref and setlinkref.

Incoderecs, inrconstrecs and instabrecs are used for copying object code, real constants and string into code, rconst and stab respectively. The instructions copied may some time contains references to rconst and stab tables. The

procedures filcinrconst and filcinstab prepare a list of such instructions and are used in inrconstrecs and instabrecs respectively.

In compiling source nodes, errors can occur. Whenever, an error occurs, the procedure error is called to produce an error message in the output file. This property of error is made use of in preparing a list of nodes whose compilation resulted in errors and marking their status as errors. This is accomplished as follows :

The variable again is used only in setline and error. Before starting scanning of a source son, setline is called which initializes again to false. In scanning, if an error is encountered, error is called, which sets again to true and marks the parent node status as errors and stores the node number of the son in the list of error nodes. Any subsequent errors do not cause the same node to be marked again, as the value of again would already have been made true.

Each node number is stored in a list of sonrec records, with a pointer to the first record in this list, stored in a linkrec record. This linkrec record is again a part of another list of linkrec records, a pointer to which is stored in the headrec record in the field errnum. The editor can display these node-numbers by following the list of linkrec records to direct the programmer to correct them.

The set of nodes to be recompiled is computed and the status of all these nodes is made new after discarding the associated compilation-information. This is accomplished by procedure changestatus in the program constructor.

Changestatus is called after completing each editing command. It is executed only if the command recently performed is one of those, which may result in the modification of the SSPF. It directly implements the rules given in Table 5.1, by a set of recursive procedures.

The program listing can be referred to for more detailed information.

A list of the procedures which occur in the program INCOMP showing their static nesting is given in Appendix C.

5.7 DEBUGGING OF THE PROGRAMS

Program testing and debugging are very crucial in any software project. If they are not carried out properly they can consume inordinate amount of time and can also result in the loss of confidence of the implementor as well as the user.

We have proceeded to test and debugg our programs in a step wise manner. During implementation and subsequent modifications, we have made use of various kinds of test outputs.

In testing the program constructor, we have taken dumps of the SSPFs created for various programs and checked to see if their structure is satisfactory (even after editing).

In testing the incremental compiler, we have obtained a dump of various compiler tables and code generated for different programs and compared them with those obtained from the original version.

During final stages, we have run a number of meaningful Pascal-S programs using the system and made sure that the results of execution are satisfactory. For each program, we have made various kinds of edits and compiled/executed them subsequently, thus achieving a reasonable level of confidence.

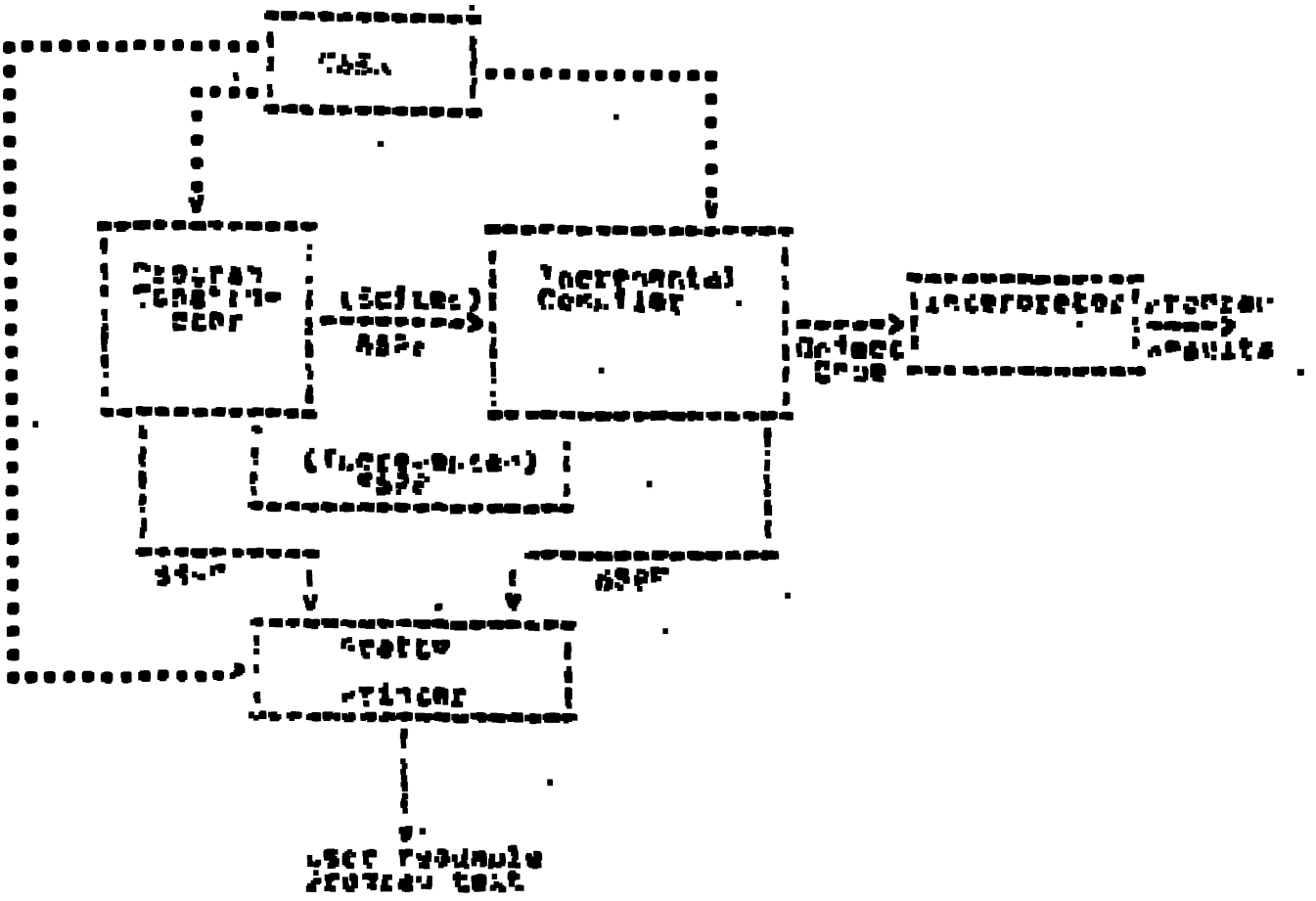
A programming system has been implemented for the programming language Pascal-S, a strict subset of Pascal. It consists of a program constructor and an incremental compiler, both working on a common interface, a structured representation of source programs called structured source program files or briefly SSPF.

The program constructor provides an editing environment which makes use of the programming language syntax. It is an editor in the sense that it is used for creating or modifying program source text. However its instructions do not operate on lines and characters but on program constructs.

The incremental compiler works on the structured representation of source program created by the program constructor. After successful compilation, it can interpret the intermediate code generated. In our programming system the program constructor and the incremental compiler are invoked alternately. The combination of the program constructor and the incremental compiler is a very economical and convenient programming tool, particularly during program development stage.

There is a pretty printing program that maps the structured object (program) in to a user readable program text with proper indentation reflecting its structure.

In the figure given below, an over view of the programming system has been presented. In this manual we have described briefly the main features of the language of implementation, Pascal-S and how to use the various components of the programming system.



Pascal-S language features.

Pascal-S is a direct subset of Pascal and Pascal-S programs is acceptable to a compiler or standard Pascal, without being subjected to the slightest change.

Pascal-S is best described by listing those features of Pascal which are not present in Pascal-S. The options in the list of the omitted features:

1. Arrays and scalar (enumerated) types
2. Variant option in records
3. Not any file data structures
4. Pointer types
5. Packing options
6. Goto and Exit statements
7. Procedures and functions being passed as parameters

Although Pascal-S does not permit file structures there are two standard text files input and output which are assumed to be implicitly declared. File input must always be included in the program heading. The inclusion of file output is optional. Operations permitted on these files are restricted to:

read/readln (on input file)
write/writeln (on output file)

4 the relative operations get/put and put/page are not allowed ;

The standard objects available in pascal-m are :
constants : true , false
types : integer , real , boolean , char
functions : sqr , sqrt , odd , chr , ord , succ , pred , round ,
trunc , sin , cos , exp , ln , sqrt , arctan , eof , eoln
procedures : get , reset , rewrite , page
In addition , we have given the ability of pascal-m to use variables

Program Construction

pasfm is a program constructor (structured, syntax directed editor) for pascal-m.
this part of the manual discusses the use of pasfm and explains our modifications to produce pascal-m
programs. it has been written from the ground up to reflect the commands/additions in the program. it assumes familiarity with the syntax of pascal-s or pascal,
and the operating environment of the microsystems. it also presupposes familiarity with the ibm text editor.
pasfm can be run in two operating system contexts (the user responses are
in upper case and explanations in all lowercase.)

OS/360:

```
FILE NAME:
```

pasfm asks for a filename. if no extension is specified the default extension ".sfm" (for structured source program files) is used.

if the file already exists on disk, in the users current path and can be read the editor responds with:

```
EXIST : VERIFY
```

otherwise the response is:

```
INPUT : NEW FILE
```

In the input mode, a program can be entered, whereas in the edit mode it allows program modification.

A simple program entry mode.

In this mode the editor requests the user to input the program in response to its queries. the program is entered against constant prompts by the editor. the dialogue is as follows:

```
INPUT : (*OS*)

  STRUCT <CONST NAME> (<CONSTANT LIST>);
    <CONST IDENT> = <CONST>;
    <CONST IDENT LIST> : <TYPE>, <TYPE>;

  VAR
  BEGIN
    <CONST IDENT> = <CONST>;
    <CONST IDENT>:=<EXP>
    <CONST>:FAIL(
  END.
    <CONST IDENT> = <CONST>;
    <CONST IDENT>:= <EXP>
```

the keywords and keysymbols such as right/left parentheses do not have to be typed. they are inserted by the editor automatically. as the editor expects a sequence of indefinite number of constant

definitions, type <CR> when you are through with the constant declaration.

To continue with the example, assuming you have typed in <CR>, the editor continues with:

```
t 10    type
            <type ident> = <type>;
            <type ident> := A
            <type> = array [MY_field..field_2] of char
t 20        <type ident> = <type>;
            <type ident>:=<CR>

v       var
v 10        <var list> : <type>;
            <var list>:=i, j,
            <type> = set of 0..9
v 20        <var list> : <type>;
            <var list> :=k,
            <type>:=.
v 30        <var list> : <type>;
            <var list> :=<CR>
```

So far, the editor has entered the following program segment:

```
c       const
c 10        max = 10;

t       type
t 10        A : array [1..10] of integer;

v       var
v 10        i, j : integer;
v 20        k : A;
```

The numbers which appear in the margin (such as v 10, v 20) are called node-numbers, and are used for editing purposes. They are not required during program entry and will be discussed later.

After this the editor query is:

Proc/Func. (P or F or <CR>) : <CR>

For entering a procedure type P, for a function type F and for none press <CR>. Since entering a procedure or a function is exactly similar to entering a program, it is omitted here for sake of brevity.

To continue, the editor responds with:

```
0       begin
0 10        <stat> :
```

This is the main program body, and a series of statements is expected. The editor has to be supplied with the type of the statement to be entered. The response to the editor query "<stat> :" can be one of the following:

Response	Statement to be entered.
as	Assignment statement
wh	While do statement
re	Repeat until statement
fo	For to statement
fod	For down to statement
ca	Case statement
co	Compound statement
pro	Procedure Call statement
if	If statement

Sample dialogues, and the statements entered in each, for each of the above responses to the query "<stat> ?", are given below.

1. ** x3 ?*

```
* 1>       <stat> : x3
* 1>       <var> := <expr>
           <var>:=x
           <expr>:=(1)+(x)
```

This enters the statement:

```
* x3       i := (1)+(x)
```

2. A* x3 ?*

```
* 1>       <stat> for
* 1>       while <cond> do <stat>
           <cond>::= i=3
* 1> 1>       <stat> : i
* 1> 1>       <var> := <expr>
              <var>::=i
              <expr>::=i+x
```

This enters the statement:

```
* 1>    while i=3 do
* 1> 1>       i := i+x
```

3. *v x1 A*

```
* 1>       <stat> : for
* 1>       repeat <stat>,,<stat> until <cond>
* 1> 1>       <stat> : x6
* 1> 1>       <var> := <expr>
              <var>::=i
              <expr>::=i+1
* 1> 1>       <stat> : <stop>
* 1>    until <cond>::=i=3
```

This enters the statement:

```
* 1>       repeat
* 1> 1>       i:=i+1
* 1>       until i=3
```

4. *v x* A*

```
* 1>       <stat>::=*
* 1>       for <var> := <expr1> to <expr2> do <stat>
           <var>::=i
           <expr1>::=1
           <expr2>::=x
* 1> 1>       <stat> : x5
* 1> 1>       <var> := <expr>
              <var>::=x+(i)
              <expr>::=1
```

This enters the statement:

```
* 1>    for i := 1 to (x) do
* 1> 1>       x:=x+(i)
```

5. ** x3 A*

```
* 1>       <stat> : for
* 1>       for <var> := <expr1> downto <expr2> do <stat>
```

```
            <var> ::=i
            <e_xpr>::=0
            <stat> : A5
            <var> := <expr>
            <var>::=<var>+1
            <expr>::=1
```

This enters the statement:

```
            for I := 1 step 1 to 10
               do A(I) := 0
```

6. ** D **

```
            <stat> : D6
            case <var> of
            <var>::=i
               <case label> : <stat>
               <case label>::=0
                  <stat> : A5
                  <var> := <expr>
                  <var>::=0
                  <expr>::=0
               <case label>::=1
                  <stat> : A5
                  <var> := <expr>
                  <var>::=0
                  <expr>::=0
               <case label>::=<var>
            end
```

This has entered the statements:

```
            case I of
               0:
                  I := 0
               1:
                  I := 0
            end
```

7. ** E **

```
            <stat> : E6
            begin
               <stat> : A5
               <var> := <expr>
               <var>::=1
               <expr>::=1
               <stat> : A5
               <var> := <expr>
               <var>::=0
               <expr>::=2
               <stat> : <EXC>
```

This enters the statements:

```
            begin
               I := 1;
               J := 2
            end
```

8. ** F **

```
            <stat> : F6
            <proc name> (<param list>)
            <proc name>::=<ID>
```

```
<term list>::=i, j, k
```

This causes the statement:

```
write (i,j,k)
```

a. ** IF **

```
1.      <stat> ; ^C
2.      IF <cond> then <stat> else <stat>
        <cond>::= (i=j) or (i=k)
3. 1.      <stat> ; ^S
4. 1. 2.      <var> := <expr>
              <var>::=i
              <expr>::=1
        else
5. 1. 4.      <stat> ; ^S
6. 1. 20      <var> := <expr>
              <var>::=i
              <expr>::=0
```

This enters the statement:

```
1.      if (i=j) or (i=k) then
2. 1.      i := 1
        else
3. 1. 2.      i := 0
```

(Once "to print the "else" part, <esc> can be pressed after the "then" part has been entered.)

a) Program editing.

Programs can be edited when the file is already on disk and the editor responds with:

```
edit : filnam.SAV
```

Also when program entry is over then the editor responds with a "*" and editing can be done.

The program resides in a file named filnam.SAV. While editing, it is first copied into a file filnam.TMP and the editing is done. After the session is over the file is copied back into filnam.SAV. So during editing, a ^C will cause a comeback to the unedited file, as the edited version is lost. In front mode, a ^C will cause all entries to be lost and is equivalent to deleting the source file.

a.1) Move numbering.

The parsed program is treated as a set of "nodes" for editing purposes. These nodes are nested one inside the other, according to the syntactic nesting structure. The outermost node is the "root" or the whole program. This contains various nodes such as constant / type / variable declarations, procedures, functions, and the program body.

Each node in the program is identified by a unique node number. The node number grows in size as the nesting gets deeper.

Consider the following program:

```
program silly ( input, output );

  const
    MAX = 10;

  var
```

```
{ 1)      a : array [1..100] of integer;
  2)        i : integer;

  3)    function f(x : integer) : boolean;

  4)      var
  5)        i : integer;
  6)        b : boolean;

  7)      begin
  8)        b := false;
  9)        for i := 1 to 100 do
 10)          a[i] := 2 * a[i-1];
 11)        f := b
 12)      end;

      begin
 13)    for i := 1 to 10 do
 14)      read(a[i]);
 15)    i := 1;
 16)    if f(i) then
 17)      writeln('yes', 'found')
      end.
```

In the above program "v 10" is the array definition, "f(x)" is the function, "f 12 b" is the "var" declaration in the function, "r 3)" is a READ statement and so on. Thus all syntactic entities in the program can be identified by the node number.

The nodes which occur inside one node are called the sons of the parent node. The node "r 3)" is a son of the node "f. 7)". "r 4)" is also a son of "f. 7)", and node "r 3) 1)" is the son of node "r 3)". Each unit in the node number (the alphanumeric characters separated by slashes) are called the son numbers. Thus node "r 1. 2)" is the 1st son number "2)" of the node "r 1)". This naming convention will be assumed hereafter.

8.2) Traversing Commands.

There exists a "current node pointer" in ASED (similar to the current line pointer in SED). This pointer points to a node number. At the beginning of the edit session it points to the "root" or the outermost node.

The current node number can be abbreviated by ".". If the current node pointer points to "n 1)" then "." can be used in commands to denote "n 1)". It can also be concatenated with more son numbers. That is if the current node is "n 1)" then ".2) 3)" will mean the node "n 1) 2) 3)".

The pointer can be moved by the following commands.

1. The "n" (goto) command.

   The goto command moves the pointer to any node. The command format is:

   n <node number>   (the "" is the editor prompt)

   The pointer to "f" will now point to "n", which is the main program node. "n 1)" will make it point to node "r 1)".

2. The "<" (up) command.

   The up command moves the pointer one level up in the nested structure. The command format is:

   <

   If the pointer is at "r 2) 1)" then typing "<" will make it point to node "r 2)". This command basically removes one unit (or son number) from the node number.

3. The ">" (enter) command.

The enter command moves the pointer one level deeper into the nesting. The command format is:

=> <son number>

if the current pointer is at "... abc then.

=>2 :

will cause the pointer to move "... abc 2" if the node exists.

4. The "<" (top) command.

The top command takes the current pointer to the root node of the program. To move to the root type:

<

The gold parameter commands that may have any effect on the current node pointer if the node or son specified does not exist. (An error message will be given). The pop command will not work if the pointer is already at the "root".

8.3) Displaying Commands.

1. The "p" (peek) command.

The peek command displays a node and its inner nested nodes on the user terminal. The format of the P command is:

=P <node number>,<depth>

The depth argument is optional. It specifies to what level of nesting the display is to be done. The default value of depth is 2.

The examples of various effects of the P command are shown below. The current node pointer is assumed to be at the "root".

```
=P
     program (a, b, c);

=P
     procedure abc(x, y, z);
       const
         xyz
       function def(p, q, r);
         begin
         end;

=P
     procedure abc(x, y, z);
       const
         xyz
       function fgh(a : integer) : boolean;
         var
```

```
begin
  end;

begin
  for  := to do
    ;
  if then
end.

procedure ATTACH ( Twoup, output );
  const
    = ;

  var
    : array [ .. ] of integer
    : integer;

  function Pulse ( : integer ) : boolean;

    var
      : ;

    begin
      :=
      for  := to do
        :=
    end;

  begin
    for  := , to  do
      ;
    if  then
  end.

procedure  ( : ; output );
  const
    = ;

  var
    : array [ .. ] of integer;
    : integer;

  function  ( : integer ) : boolean;

    var
      : ;

    begin
      := ;
      for  := , to  do
        := ;
      := ;
    end;

  begin
    for  := , to  do
      ;
    if  then
  end.
```

The P command performs a "space" before displaying. So the current node pointer is moved to the node displayed.

2. The "*" (next) command.

The next command displays the current node pointer and the sons of the current node on the users terminal. If you are at the node 4 in which, suppose, is a compound statement with 3 sons, the next command shows:

```
*
 Current node :  10
Sons of this node : 11  2u  3.
*
```

If the current node pointer is at the root, no node number is displayed as the "Current node:", as the node number of the root is null.

3. The "F" (Find) command.

The find command accepts a string and a nodenumber and searches the subtree rooted at this node for an occurrence of this string. When a node whose source has an occurrence of the string is found then the current node pointer is made to point to this node.Typing F again and again causes the same string to be searched for in the rest of the subtree.If no occurrence is found it reports failure and the current node pointer will be pointing at the node specified in the command.The format of the command is :
*F<string><esc><nodenumber>

An example is shown below.

```
*P.,5
    program SEARCH ( input, output );
C     const
C 10      MAX = 10;

V     var
V 10      A : array [1..MAX] of integer;
V 20      J,I : integer;

F10       function FIND ( X : integer ) : boolean;

F10 V       var
F10 V 10        J : integer;
F10 V 20        FND : boolean;

F10 .       begin
F10 . 10        FND := false;
F10 . 20        for J := 1 to MAX do
F10 . 30 10       I := J;
F10 . 30        FIND := FND
            end;

.       begin
. 10        for I := 1 to MAX do
. 10 10       READ (A[I]);
. 20        READ (X);
. 30        if FIND(X) then
. 30 10       WRITELN ('Found')
        end.

*F<esc>.
F10       function FIND ( X : integer ) : boolean;

F10 V       var
F10 V 10        J : integer;
F10 V 20        FND : boolean;
```

```
[illegible]
[illegible]
[illegible]
```

3.5) Editing Commands.

1. The "d" (delete) command

The delete command deletes one node from the program. The format of the delete command is:

d) <seq number>

For the d command, the seq number has to be specified, and it will operate on the seq of the current node. Thus to delete node #3, the commands will be:

```
f3 (v
*) d)
Node 3 deleted.
*
```

Some nodes cannot be deleted due to syntactic considerations. For example the statement part of a while to statement cannot be deleted. For example:

```
*f3 (v
3 ).      while x=0 do
3 ) )       x := x+1
*d)
Cannot delete.
*
```

For deleting statements in such situations use the "e" command, which is given later.

The I (insert) command.

The i command inserts a seq in the currently positioned node. If the current node pointer is pointing to an "if" then:

```
*i3)
```

will create one node "3 e" for "else" provided it can be created, and prompt for similar to that coding program entry will be done to create the new nodes.

The i command can be used to insert statements into compound statements, or repeat until statements. It also inserts procedures or functions when the current node are the seq number are given appropriately. To insert declarations first go to the appropriate declaration node in the declaration node is not displayed it no definitions are contained in it, but it exists all the same). The following example is pertinent to entering declarations.

```
*f...)
     [illegible]

*i...)
     [illegible]

*i...)
     [illegible]
     end;

*
[illegible] of this node [illegible]

*i)
     <type ident> = <type>
     <type ident>:=A.F
     <type>:=array [1..10] of integer;
[illegible] inserted.
*
```

```
program not complete!

    type
      ... = array [1..1] of integer;

    var
      a : arrayer;

    begin
      i := 1
    end.
```

The i command will not work if the son number specified already exists, unlike the big editor which will appropriately increment the line number in such cases. Each has to be given the exact number of the son to be inserted. This command has no effect on the current node pointer.

The "r" (replace) command.

The r command replaces a node. This command takes a son number as an argument and looks on that son of the current node (like 2 and 3). However unlike the 2 and 3 commands, this can replace any node with no exceptions.

Suppose we have a program segment:

```
      while a<>5 do
        a := a+1
```

And we want to make changes to:

```
      while a<>5 do
        a := a+2
```

Then we can do the following:

```
*p , a
      while a<>5 or
        a := a+1
*p 3,3
3 while a<>5 or , selected for replacement.
*r 3
=
new replaces.
*p 3
      while a<>5 do
        a := a+1
*p 3,3
3 while a<>5 do , selected for replacement.
*r 3
        <stmt> ; a5
        <var>:= <expr>
        <var>:=a
        <expr>:=a+2
new replaces.
```

The replace command can be used to replace any node anywhere in the program, including procedures, functions, declarations. All the examples of modifications given for illustration here, this command does not affect the current node pointer.

The "d" (delete) command.

The d command deletes the source associated with a node. The command takes a son number as an argument and looks on that son of the current node (like 2 and 3 commands). But like the 3 command, it can alter any node with no exception.

Suppose we have the program segment

```
* 10     FOR I := 1 TO
* 10 1>   IF CH[I] <= 'Z' THEN CH[I] := CHR(ORD(CH[I])+32);
```

and we want to change it to

```
* 10     FOR I := 1 TO
* 10 1>   IF CH[I] <= 'Z' THEN CH[I] := CHR(ORD(CH[I])+32);
```

then using a command we can do the following

```
*pv 1>
* 10     FOR I := 1 TO
* 10 1>   IF CH[I] <= 'Z' THEN CH[I] := CHR(ORD(CH[I])+32);
*>1>
*P
* 10 1>   IF CH[I] <= 'Z' THEN CH[I] := CHR(ORD(CH[I])+32);
*aEAPn
* 10 1>   FOR
```

At this stage the source can be altered by deleting , inserting, or replacing characters by typing the following commands.The cursor is initially pointing at the first character.It can be set to point at any character and make it the current character.

Cursor	Skip the current character
^	Delete the current character
^^	Delete the character before current character
^	Delete from the after here after displaying the rest of the source
I	Insert a string of characters at the current character position.The string ends with <ESC>.
R	Display the source and recycle.
R	Replace the current character by a string. The string ends with <ESC>.
Sch	Search for an occurrence of ch in the string and make it the current character.
-Sch	Search for an occurrence of ch to the left of the current character position.
C	Convert the current character from lower case to upper case or vice versa and display it.The character position also moves once to the right.
-B	(Back Space) move one character position towards left.
^^	Skip to the end of the source.
^^^	Skip to the beginning of the source.

The alter can be exited by typing <ESC> also.

The "S" (substitute) command.

This command substitutes one string (called source string) , another string (called substitution string) in the subtree whose root is specified in the command.It has the format :

/<source string><ESC><test string><ESC><occurrence>

An example is given below.

```
        PROCEDURE SUBSTR ( STR:PTR; POS,LEN:INTEGER);
        VAR
          I,J,K,L,M,N : INTEGER ;
          FOUND : BOOLEAN ;
        BEGIN
          L := LENGTH(STR^) ;
          M := LEN ;
          N := 1 ;
          FOR I := POS TO POS+LEN-1 DO
            IF STR^[I] = CH THEN
              BEGIN
                FOUND := TRUE ;
                FOR J := I+1 TO I+LEN-1 DO
                  IF STR^[J] <> CH2[N] THEN FOUND := FALSE ;
                IF FOUND THEN
                  BEGIN
                    FOR K := I+LEN TO L DO
                      STR^[K-LEN+CH3.LENGTH] := STR^[K] ;
                    L := L-LEN+CH3.LENGTH ;
                    FOR K := 1 TO CH3.LENGTH DO
                      STR^[I+K-1] := CH3[K] ;
                    M := M-1 ;
                    IF M = 0 THEN EXIT(SUBSTR) ;
                    N := 1 ;
                  END
                ELSE N := 1 ;
              END ;
        END ;
```

```
Current node is 210
Node of this node : <data part> 0 1 2 3
Searching for <source string>.
<source string>
Search fails.
```

If no occurrence of the <source string> is found it reports failure as shown above.

After execution of f command the current node pointer will be pointing to the node specified in the command.

5.6) Node Editing Commands.

1. Temporary Registers.

The editor has 26 temporary storage registers. One node and its secondary and later nested nodes can be stored in a register. The registers have names a, ab, to z. These registers can be used in the transfer and copy commands for easy movement of nodes in the program.

2. The "t" (transfer) command.

The transfer command can move a node from one point in the program to another, or move a node from one point in the program to a register or vice-versa.

The command format for the t command is:

```
* <destination node no.>=<source node no.>
        or
* a=<source node no.>      a = a to z
        or
*t<destination node no.>=a
```

The t command deletes the node from the source and inserts it at the destination. It is constrained by the fact that some node may not be deleted and some nodes may not be inserted. It never overwrites a node, that is it will not insert a node if a node already exists at that point.

3. The "c" (copy) command.

The copy command is similar to the t command except that it makes an identical copy of the source node at the destination. Thus the source node is preserved and not deleted. Moreover the copy command can overwrite, that is if the destination node already exist it will get superseded. This might be useful at times, but quite dangerous too.

4. The "f" (find) command.

The find command has a format:

```
f <node number>
```

It is a very similar function. If the node specified is a record node then it will be converted to a command statement. This is useful for converting a record until to a utility in source out.

5. The "w" (write) command.

The write command saves all the edits and insertions upto that point into the source file "filename.jbr". It is advisable to use the command to save the file occasionally during edit sessions to safeguard against system crashes and inadvertant ^Cs. There is no undo save feature.

F.7) Exit Commands.

1. The "E" (exit) command.

The exit command terminates the edit session, saves the file and returns you to the monitor. It is the proper way to end the session, and causes all the editor portions intact. Terminating an edit session with a ^C will cause you to come back to the same file.

2. The "EC" (compress and exit) command.

This command is exactly the same as the "E" command, but it compresses the file before it exits. During an edit session a lot of "garbage" is generated in the file. The EC command cleans out all the unwanted matter from the file and makes it more compact.

G) Miscellaneous

1. The transfer and copy commands should be used with discretion. Although the editor checks for errors due to nodes written, inserted where they do not belong, the checks are not too tight. Hence it is possible to introduce syntax errors with the use of these commands. This should be avoided.

2. The "allowed nodes" for all the nodes in the program are given below. The table shows all the sons of each possible node in the program. The sons which have 3 or 4 character alphabetic names are the allowed ones as these names do not appear in the context of the program. However it is more advisable, to check on the proper numbers of the nodes when in doubt by the "?" command.

Node Type	Sons
program (...	name, Defs, C, x, y, ...
const	id, Zv ..
const defn.	Name, Const
(e.g. c = 1)	id, Zv ..
type	id, Zv ..
type defn.	Name, Type
(e.g. r = ...)	id, Zv...
var	id, Zv ..
var defn.	First, Type
(e.g. i : int)	
procedure (	Name, Defs, C, x, y, ...
function (i:	Name, Defs, Type, C, x, y, ...
parameter	id, Zv ..
repeat...until	Zv, Zv .. , cond
while...do	cond, Zv
if then else	Cond, Zv, Zv
for := to	id, expr, expr, Zv
for := downto	id, expr, expr, Zv
procedure call	Name, Zv...
case of	expr, Zv, Zv, ..
each case (range-list	
(i.e. son of a case stat)	Zv, Zv

3. Input entering record structures in type declarations type RECORD and then press <return>. The editor will perform proper indentation for the fields. Press <return> after entering each field. The entry process will stop after you type "end".

4. The editor does no context sensitive checks. Thus wrong identifier names, undeclared names and so on will pass undetected. But while it does check the character strings you present it with when it asks for expressions or variables or parameter declarations. No error messages are given during input. If you make errors in a portion which has been listened there is nothing you can do as long as you are in the input mode. When asked next to edit, only then you can replace the bad strings.

[illegible] will not however accept a null string. That is if it asks for an expression just a <CR> will not satisfy. It keeps on waiting and ignoring <CR>s until you type some characters before the <CR>. However a <ESC> will force the acceptance of a null string.

5. If the editor is being used to modify an already compiled program that has errors, then the editor will display a list of [illegible] as soon as the editor is entered. The user can make use of this list to make corrections to the program. All the [illegible] displayed may not actually contain errors. They may be marked erroneous as a result some other notes until erroneous. For example a variable name might have been declared wrongly and this causes an error and the corresponding notes are listed. All these notes which have an occurrence of this name may also be listed, as these also cause errors. The user has to use his common sense in making use of this list to make corrections.

As each note listed is modified its [illegible] is deleted from this list. At any time the remaining unmodified notes can be seen by the "n" (list) command, which has the following format :

*n

6. A brief note on various commands provided by the program constructor can be obtained by the "?" (help) command. It has the following format :

*?

D) Error messages.

1. % Command error.

A command string that [illegible] cannot recognize was found.

2. % (illegal) char in filename.

Characters which are not alphanumeric or "." were found in the file spec.

3. % File extension has to be .PAS

The file spec specifies another extension that was not .PAS.

4. % Error in octal number.

A [illegible] has not specified properly, i.e. a digit was missed out or illegal characters were used.

5. % Note not available.

No note was given as argument to the *, C, T, ? or > command.

6. % (illegal) char in cmd line.

Characters which are not alphanumeric or . , = > appeared in the command line.

7. % [illegible].

The [illegible] in a *, >, or C command does not exist in the program.

8. % Not in range.

The [illegible] in the C, < or > command does not exist in the program.

9. % Node already exists.

A node specified in a J command or specified as a destination node in a T command, exists and cannot be overwritten.

10. % Can't delete.

The delete command failed on a node, i.e. an attempt was made to delete a node which cannot be deleted due to syntax considerations. Use the X command in such cases.

11. % Can't insert.

Attempt to insert a node at a point where it cannot be inserted due to syntax considerations.

12. % Transfer node not compatible.

Attempt to transfer a node by T or C command to a point in the program where it cannot reside due to syntax considerations (e.g., transferring a var declaration into a compound statement).

13. % Register to register transfer illegal.

Pascal does not allow contents of one register to be transferred to another register.

14. % Illegal node number.

A node number specified by an S command is not valid in this context.

15. % Equal sign (=) missing.

The equal sign (=) in the T or C command is missing. Syntax error in T or C command.

16. % Illegal register name.

Anything apart from A-H used as a register name.

17. % Transfer point not found.

The node specified as the destination in the T or C command cannot exist in the program.

18. % Register or contents missing.

Attempt to transfer / copy from an empty register.

19. % Register or is not empty.

Attempt to overwrite a register by a T command. (A C command can overwrite but a T cannot.)

20. % Register to register copy illegal.

Pascal does not allow copying from one register to another.

21. % Illegal in insert mode.

Attempt to use registers while entering statements in input mode.

22. % Too many single statements.

Attempt to use 9 or more single statements.

23. % Illegal character in control string.

[illegible] failed. The node specified is not a repeat until sensor.

24. A search fails.

The [illegible] command fails to locate an occurrence of the search specified.

Incremental Compiler

This is an incremental compiler for Pascal-s programming language.It works directly on the DAG created by the program constructor. It is invoked by the following operating system command :

[illegible]

The compiler asks for a file specification (filename and switches ,if any).It has the format :

<filename>.<ext>/<switches>

The default extension is ".SAS" for structured source files.The switches are :

/LS	(default)	Produces a listing of all the nodes [illegible] in the DAG , in turn.
/NL	(default)	Suppresses the listing.
/E	(default)	Interprets the intermediate code generated after successful compilation.
/NE		Does not interpret the generated code.

If an error occurs in compilation the system displays the following message.

? ERROR ENCOUNTERED

If compilation is successful , then if /E switch is on then the compiler proceeds to interpret the generated code after displaying the message

***** Execution Starts *****

and the results of execution can be found in the file OUTPUT.

If /NE switch is on then the system just displays the following message.

[illegible] END OF

After compilation the system marks the status of all those nodes whose compilation resulted in errors as "ERROR" and makes a list consisting of the addresses of all such nodes and stores it in the root node of the source tree.The program constructor (editor) makes use of this list to direct the programmer to correct all such erroneous nodes by displaying this list either automatically , when it is first entered , or on demand.

The system aborts compilation after encountering an irrecoverable error in the DAG and displays an appropriate error message.

Several messages can be given , which are listed below with a brief note about each , how it can occur.

1. ? Compiler table for identifiers is too small.

   Symbol table (ST) size in the compiler should be larger.

2. ? Compiler table for procedures is too small.

   Block table (BT) in the compiler should be larger.

3. ? Compiler table for reals is too small.

Real constant table (RCONST) in the compiler should be larger.

4. ? Compiler table for arrays is too small.

Array table (ARAB) in the compiler should be larger.

5. ? Compiler table for levels is too small.

Maximum nesting allowed in the programs is 7.

6. ? Compiler table for code is too small.

Code table (CODE) in the compiler should be larger.

7. ? Compiler table for strings is too small.

Total length of all the strings in the program can't exceed 1000 characters.

8. ? Not positioned on ------ record when expected.

Last node processed (or being processed) was -------

at address --------.

The compiler expected ------ node in the SSPF but found something else. Most commonly this message can appear if the compiler is given a file name that is in the directory.

9. ? Source incomplete.

The scanner has been asked to scan beyond the end of last source record of the current node.

10. ? Maximum real constants in one node can't be greater than -----

This error occurs for the node number ----.

11. ? Maximum number of strings in one node can't be greater than -----

This error occurs for the node number ----.

12. ? Node number does not have ---- reconstrecs as expected.

13. ? Node number does not have ---- stasrecs as expected.

14. ? Statement node not found when expected.

In the listing file the compiler writes only error numbers and keywords corresponding to these numbers for brevity.The following is an exhaustive list of all the error message numbers along with a note as to what they are:

1. The [illegible] identifier is declared more than once in the same scope.

2. An identifier is expected.

3. [illegible] expected.

4. A [illegible] is expected.

5. A colon is not found in declaration when [illegible] is followed by type.

6. A type other than the [illegible] symbol is incorrectly used.The compiler skips this and possibly several other following symbols.

7. In a formal parameter list each section must begin with an identifier or the symbol VAR, depending on whether the parameter is a value or variable parameter.

8. Type symbol or expected.

9. An opening parenthesis is expected.

10. A type identifier must begin with an identifier, the symbol ARRAY or the symbol RECORD.

11. An opening bracket is expected (.

12. A closing bracket is expected).

13. The symbol .. is expected (no blank between the dots).

14. A declaration is expected.

15. The result of a function must be of type integer, real, boolean, or char.

16. An assignment is expected. The symbol := is used in assignment statements only, but not in declarations.

17. The expression following the symbol IF, WHILE, or UNTIL must be of type boolean.

18. The control variable following the symbol FOR must be of type integer, char or boolean.

19. The expressions which specify the initial and final values of the control variable in a FOR statement must be of the same type as the control variable.

20. The parameter 'output' must be included in the program heading.

21. The indicated number is too large. The maximum number of digits is 14. The absolute value must not exceed 1.0**123 (on the implementation).

22. Loop index cannot be a VAR parameter.

23. The expression following the symbol CASE must be of type integer, char or boolean. (In the latter case, an IF statement is recommended.)

24. The designated character is not acceptable.

25. In a constant definition, the equal sign must be followed by a constant. If an identifier is used it must denote a constant.

26. The type of an index expression must be identical to the index type specified in the array declaration.

27. In an array declaration the lower bound must not exceed the upper bound, and must be within a permissible range of values (less than 2**31). The index types must be identical, either integers, boolean values or characters. Real numbers are not acceptable.

28. Every indexed variable must be declared as an array.

29. A type identifier is expected here.

30. This type is not defined. (Recursive type definitions are not allowed.)

31. Every variable for a field selector must be declared as record.

32. The operands of the operators not, and, and or must be of type boolean.

33. The specified type of this arithmetic expression is illegal. Note also that entire arrays cannot occur as operands to arithmetic or logical operators.

34. Operands of div and mod must be of type integer.

35. The types of the comparands are incompatible. They must be identical, except if one compared is of type integer and the other of type real. Arrays must be compared element by element.

36. The types of corresponding actual and formal parameters must be identical. An exception is made if the formal parameter is a value parameter of type real, then the actual parameter may also be of type integer.

37. A variable is expected.

38. A string must contain at least one character.

39. The number of actual parameters must be equal to the number of formal parameters.

40. The parameters of procedure read must be of type char, integer, real or boolean.

41. If a statement has a form write(X:Y:Z) , then Y must be of type real.

42. If a statement has a form write(X, or write(X:Y:) , then Y and Z must be expressions of integer.

43. No type or procedure may occur as part of an expression.

44. A statement cannot begin with a type or a function identifier. An exception is the assignment of a result value to a function. In this case it must be part of the function body.

45. In an assignment x := y , the types of the variable x and expression y must be identical. An exception is the case when x is real, then y may also be of type integer.

46. Every case label must be a constant of the same type as the expression in the case clause.

47. The formatted argument of the standard function is of illegal type.

48. The program requires too much storage.

49. A constant cannot begin with an indicated symbol.

50. The symbol is is expected. E.g. a space between : and = .

51. Too many nested scopes.

52. The symbol that begins not, an identifier, a constant, the symbol not, or the left parenthesis.

53. Too many digits in a string or constant.

54. The forward declared identifier is not a procedure identifier.

prettyp is a pretty printing program that uses the SAIL given to it in to a user readable (nroff) text with proper indentation reflecting its structure.

An use prettyp as the following :

```
@prettyp
File : file.sail

exit;
.
```

The extension defaults to ".sail" if not specified. It creates a file called filename.prp containing the program text. The node numbers of the ".sail" file appear in the ".prp" file as comments.

CHAPTER 7

CONCLUDING REMARKS

We have implemented a programming environment that is aware of the structure of the source programs. The importance of such an environment need not be overemphasized.

We have started with an existing syntax oriented editor for Pascal-S and an incremental compiler for the same programming language and modified these two to work on a common interface, thus deriving a powerful programming tool.

Present system can be used as an instructional tool for students taking a first course in programming. Such a system provides a healthy and uniform-programming environment for the students and also makes it unnecessary for them to know the syntactic details of the programming language thoroughly.

We feel that such a programming system can be more useful if extended to the full set, programming language. The nature of the compiler we have implemented is such that the object code produced makes a number of references to the compiler tables. Much work can be saved if object code is made independent of the compiler tables.

There is scope for improving the program constructor. Error information can be communicated to the user more effectively instead of simply displaying a list of node-

numbers which have errors. Probably along with the node numbers, displaying the errors in the source code may help the programmer. Also the present version of the program constructor does no context sensitive checks and the burden of the compiler can be further reduced by incorporating such a feature in the program constructor itself.

The Random Access file package makes it necessary for the file to be copied into a temporary file and after compilation, again the temporary file to be copied back into the original file. By suitably altering this package, this duplication of files can be avoided, thus saving a lot of CPU time as well as response time.

REFERENCES

[WIR 75] Wirth, N., 'PASCAL-S: A subset and its implementation', Tech. Report, ETH, Zurich 1975.

[JAJ 78] Jajoo, B.H., and Sahasrabuddhe, H.V., 'Flowtrees - a graphic technique to represent structured programs', Tech. Report No. TRCS-78-001, C.C., I.I.T. Kanpur, Kanpur, 1978.

[HAB 79] Habuman, A.N., 'Tools for software systems construction', in proceedings of a Workshop on Software development tools (1979), Pingree Park, Colorodo, pp. 10-21.

[JAJ 80] Jajoo, B.H., 'A programming system based on program structure', Ph.D. Thesis, August, 1980, Computer Science Programme, I.I.T. Kanpur.

[PAR 81] Partha Dasgupta, 'A syntax oriented editor for Pascal-S', M.Tech. thesis, July 1981, Computer Science Programme, I.I.T. Kanpur.

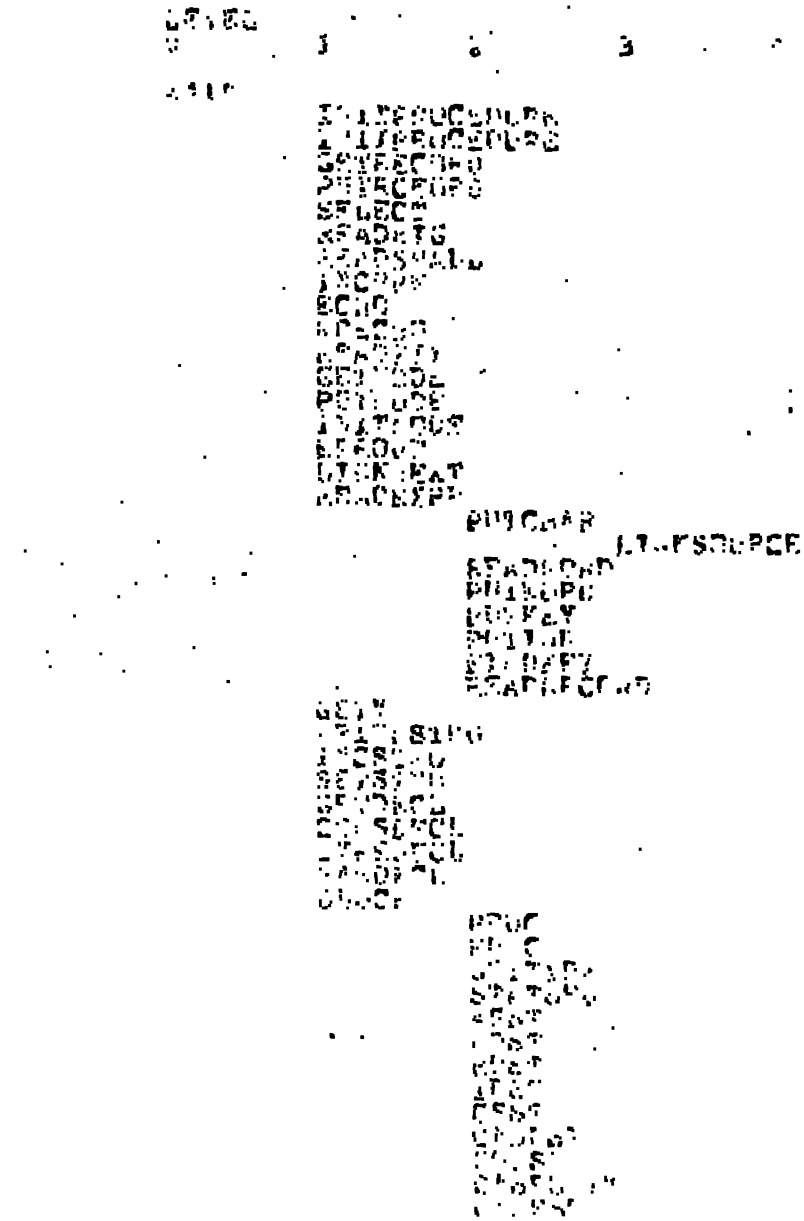
[TEI 81] Teitelbaum, T., Reps. T., 'The cornell program synthesizer : A syntax-directed programming environment', Comm. ACM 24 (1981), pp. 563-573.

APPENDIX A

[illegible] LISTING VARIOUS COBOL TYPES USING ALTER VERB IN COMPILED-TIME VARIATION

[illegible]	[illegible] [illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

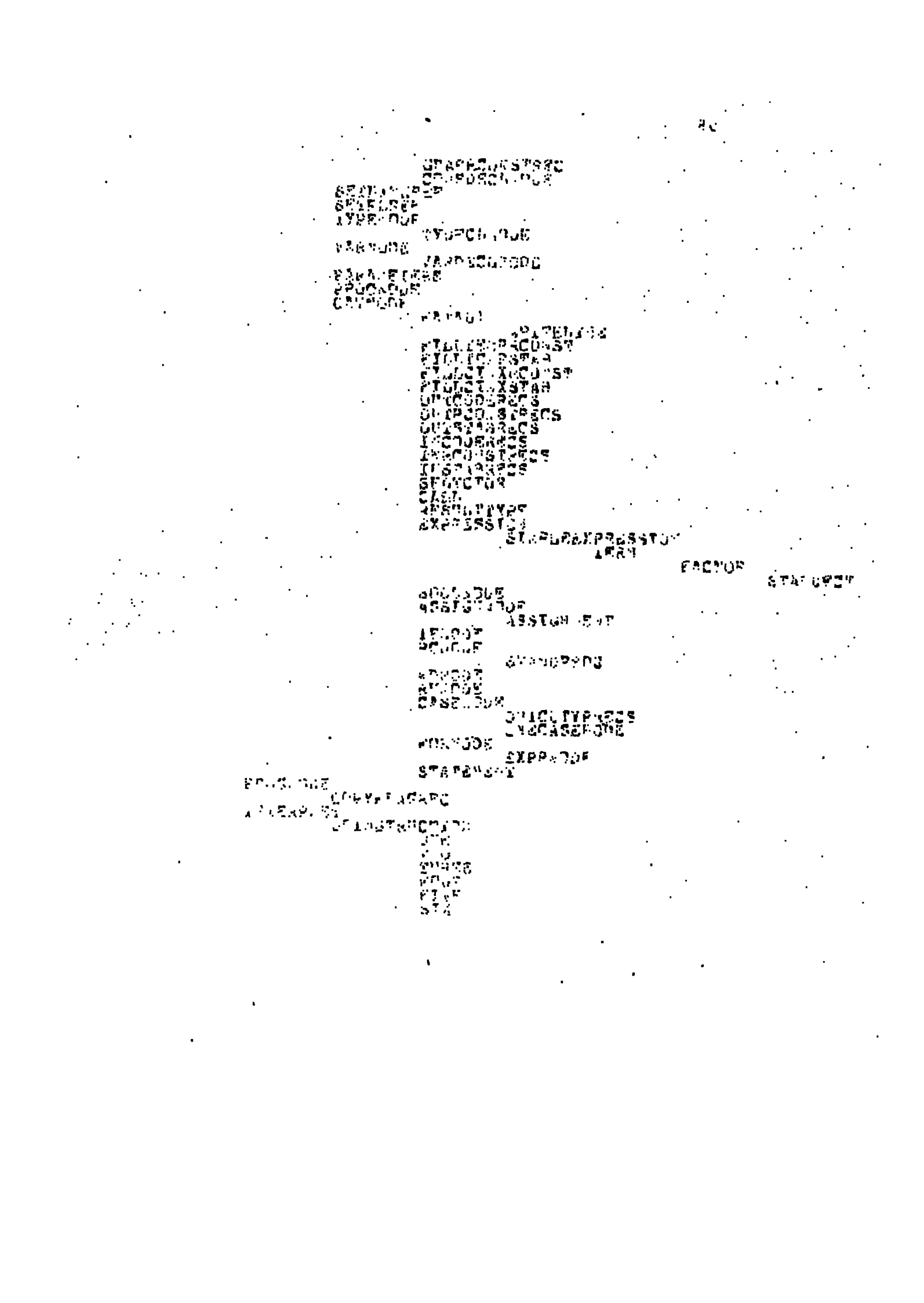
LIST OF PROCEDURES USED IN THE PROGRAM PARSER, SHOWING AREA
STATIC ALLOCATED.



APPENDIX C

S[illegible] LISTING OF THE PROCEDURES USED IN THE PROGRAM INCLUDE, SHOWING WHETR
USED
CALL

```
1       2       3       4       5       6       7
        DRAWCT
        DRAWRECTS
        DRAWSECTS
        DRAWSQ
        MEASURETIC
        GRAPHINVAR
        GRAPHICS
        GRAPHICSINIT
        INITIALIZE
        PLACEBLOCKSUSED
        PRINTRECORD
        QUESTIONS
        QUESTIONSUB
        SELECTSQ
        SETUPBLOCKSUSED
        SETUPSEC
        TRACE
                EXTLINE
ERROR
                GETSUPPARMS
INSYSUB
                READSCALE
                TOUCHSCALE
ENTER
ENTERBOARDAY
ENTERSECTOR
EXITTA
EXITTA
PRINTPLAYFLD
SECTOR
SETUPTRACK
SQUARE
                CHECK
                CHOOSETOPLOC
                COLOR
                CREATEV
                ARRELEA
                STRELEA
                TRACE
                WRITEVARIABLE
                COLSTART
                IVL
                        CHOOSETOP
                CONVERSEC
                CONVTARSEC
                CONVTOSEC
                DRAWTARSEC
                DRAWTOSEC
                INITTARSEC
                INITTOSEC
                QUESTIONSEC
                        QUESTOSEC
                        QUESTARSEC
```



APPENDIX A

...ascal-S syntax in B.N.F (backus - naur form).
Syntactic constructs are denoted by english words enclosed in angular brackets "<" and ">".these words also describe the nature and meaning of the constructs.
Possible repetitions of a construct is indicated by an asterisk "*" (0 or more times) or an exclamation mark "!" (1 or more times).
If a sequence of constructs to be repeated consist of more than one element it is enclosed by the meta-brackets "(" and ")" which imply a repetition of 0 or more times.
The basic vocabulary consists of basic symbols classified as to letters digits and special symbols.

<letter>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>::=0|1|2|3|4|5|6|7|8|9
<special symbol>::=+|-|*|/|=|,|.|<|>|<=|>=|(|)|[|]|:=|(.|.)|'|:|;|^|
and|or|not|div|mod|if|then|else|case|of|repeat|
until|while|do|for|to|downto|begin|end|const|
var|type|array|record|function|procedure|program

The construct
(* < any sequence of characters not containing "*)" > *)
may be inserted between any two identifiers ,numbers or special symbols.
It is called a comment and may be removed from the program text without altering its meaning.
The symbols "::=" , "|", "," and "*" are meta-symbols and used to describe the language.

<program>::=<program heading> <block>.
<program heading>::=program <identifier> (<idlist>) ;
<idlist>::=<identifier>! , <identifier> /
<identifier>::=<letter> <letter or digit>*
<letter or digit>::=<letter>.<digit>
<block>::=<constant definition part> <type definition part> <variable
declaration part> <procedure and function declaration part>
<statement part>
<constant definition part>::=<empty>!const <constant definition> ;
<constant definition> ; *
<constant definition>::=<identifier> = <constant>
<constant>::=<unsigned number>!<sign> <unsigned number>!
<constant identifier>!<sign> <constant identifier>!
<char constant>
<unsigned number>::=<unsigned integer>!<unsigned real>
<unsigned integer>::=<digit>!
<unsigned real>::=<unsigned integer> . <digit>! .
<unsigned integer> . <digit>! & <scale factor> !
<unsigned integer> & <scale factor>
<scale factor>::=<digit>! !<sign> <digit>!
<sign>::=+ ! -
<constant identifier>::=<identifier>
<char constant>::="<character>
<type definition part>::=<empty>! type <type definition> ;
! <type definition> ! /
<type definition>::=<identifier> = <type>
<type>::=<simple type>!<structured type>
<simple type>::=<type identifier>
<type identifier>::=<identifier>
<structured type>::=<arraytype>!<record type>
<arraytype>::= array [<indextype> ! of <component type>
<index type>::=<constant>..<constant>
<component type>::=<type>
<record type>::= record <field list> end
<field list>::=<record section> ! ; <record section> !

```
<record section>::=<field identifier> { , <field identifier> } : <type>
<variable declaration part>::=<empty>|
                              var <variable declaration> ;
                                  { <variable declaration> ; }
<variable declaration>::=<identifier> { , <identifier> } : <type>
<procedure and function declaration part>::={ <procedure or function
                                                declaration> ; }
<procedure or function declaration>::=<procedure declaration>|
                                      <function declaration>
<procedure declaration>::=<procedure heading> <block>
<procedure heading>::= procedure <identifier> ;|
 procedure <identifier> ( <formal parameter section>
                        { ; <formal parameter section> } ) ;
<function declaration>::= function <identifier> ; <result type> ;|
                          function <identifier> ( <formal parameter section>
                          { ; <formal parameter section> } ) : <result type> ;
<formal parameter section>::=<parameter group>| var <parameter group>
<parameter group>::=<identifier> { , <identifier> } : <type identifier>
<result type>::=<type identifier>
<statement part>::=<compound statement>
<compound statement>::= begin <statement> { ; <statement> } end
<statement>::=<simple statement>|<structured statement>
<simple statement>::=<assignment statement>|<procedure statement>
<assignment statement>::=<variable> := <expression>|
                         <function identifier> := <expression>
<expression>::=<simple expression>|
               <simple expression> <relational operator>
                                   <simple expression>
<simple expression>::=<term>|
                      <simple expression> <adding operator> <term>|
                      <sign> <term>
<term>::=<factor>|<term> <multiplying operator> <factor>
<factor>::=<variable>|<unsigned constant>|<function designator>|
          ( <expression> )| not <factor>
<relational operator>::= = | <> | < | > | >= | <= | in
<multiplying operator>::= * | / | div | mod | and
<adding operator>::= + | - | or
<function designator>::=<function identifier>|
                       <function identifier> ( <actual parameter>
                       { , <actual parameter> } )
<function identifier>::=<identifier>
<procedure statement>::=<procedure identifier>|
                        <procedure identifier> ( <actual parameter>
                                               { , <actual parameter> } )
<procedure identifier>::=<identifier>
<actual parameter>::=<expression>|<variable>
<structured statement>::=<compound statement>|<conditional statement>|
                         <repetitive statement>
<conditional statement>::=<if statement>|<case statement>
<if statement>::= if <expression> then <statement>|
                  if <expression> then <statement> else <statement>
<case statement>::= case <expression> of
                        <case list element> { ; <case list element> } end
<case list element>::=<case label list> : <statement>
<case list element>::=<case label> { , <case label> }
<repetitive statement>::=<while statement>|<repeat statement>|
                         <for statement>
<while statement>::= while <expression> do <statement>
<repeat statement>::= repeat <statement> { ; <statement> }
                      until <expression>
<for statement>::= for <control variable> := <forlist> do <statement>
<control variable>::=<identifier>
<forlist>::=<initial value> to <final value>|
            <initial value> downto <final value>
<initial value>::=<expression>
<final value>::=<expression>
```